CUBRID 2008 R4.0 Help

Table of Contents

Introduction to Manual

Manual Contents

The contents of the CUBRID Database Management System (CUBRID DBMS) product manual are as follows:

- Introduction to CUBRID : This chapter provides a description of the structure and characteristics of the CUBRID DBMS.
- Getting Started with CUBRID : The "Getting Started with CUBRID" provides users with a brief explanation on what to do when first starting CUBRID. The chapter contains information on new features added to CUBRID, on how to install and execute the system, and provides a simple guide on how to use the CSQL Interpreter and CUBRID Manager. The chapter also includes examples of how to write application programs using JDBC, PHP, ODBC, CCI, etc.
- CSQL Interpreter : CSQL is an application that allows you to use SQL statements through a command-driven interface. This chapter explains how to use the CSQL Interpreter and associated commands.
- CUBRID SQL Guide : This chapter describes SQL syntaxes such as data types, functions and operators, data retrieval or table manipulation. The chapter also provides SQL syntaxes used for indexes, triggers, partitioning, serial and user information changes, etc.
- Administrator's Guide : This chapter provides instructions on how to create, drop, back up, restore and migrate a database, and executing CUBRID HA functionality. Also it includes instructions on how to use CUBRID utilities, which starts and stops the Server, Broker and CUBRID Manager servers, etc.
- Performance Tuning : The "Performance Tuning" chapter provides instructions on setting system parameters that may influence the performance. This chapter provides information on how to use the configuration file for the Server and Broker, and describes the meaning of each parameter.
- API Reference : This chapter provides information on JDBC API, ODBC API, OLE DB API, PHP API, and CCI API.

Manual Conventions

The following table provides conventions on definitions used in the CUBRID Database Management System product manual to identify "statements," "commands" and "reference within texts."

Convention	Description	Example
Italics	Italics type is used to show the variable names.	*persistent*: stringVariableName
Boldface	**Boldface** type is used for names such as the member function name, class name, constants, CUBRID keyword or names such as other required characters.	**fetch** ( ) member function class **odb_User**
Constant Width	Constant Width type is used to show segments of code example or describes a command's execution and results.	csql database_name
UPPER-CASE	UPPER-CASE is used to show the CUBRID keyword (see **Boldface**).	**SELECT**
Single Quotes (' ')	Single quotes (' ') are used with braces and brackets, and shows the necessary sections of a syntax. Single quotes are also used to enclose strings.	{'{'const_list'}'}
Brackets ([ ])	Brackets ([ ]) indicate optional parameters or keywords.	[ONLY]

<table>
<tr><td>Underline(_)</td><td>Underline (_) indicates a default keyword if no keyword is specified.</td><td>[DISTINCT|UNIQUE|ALL]</td></tr>
<tr><td>Vertical bar(|)</td><td>Vertical bar (|) indicates that one or another option can be specified.</td><td>[COLUMN|ATTRIBUTE]</td></tr>
<tr><td>Braces around parameters({ })</td><td>Braces around parameters indicate that one of those parameters must be specified in a statement syntax.</td><td>{2, 4, 6}</td></tr>
<tr><td>Braces with ellipsis({ }...)</td><td>Braces before an ellipsis indicate that a parameter can be repeated.</td><td>{, class_name}...</td></tr>
<tr><td>Angle brackets(< >)</td><td>Angle brackets indicate a single key or a series of key strokes.</td><td><Ctrl+n></td></tr>
</table>

Introduction to CUBRID

Introduction to CUBRID

This chapter explains the architecture and features of CUBRID. CUBRID is an object-relational database management system (DBMS) consisting of the Database Server, the Broker, and the CUBRID Manager. It is optimized for Internet data services, and provides various user-friendly features.

This chapter covers the following topics:

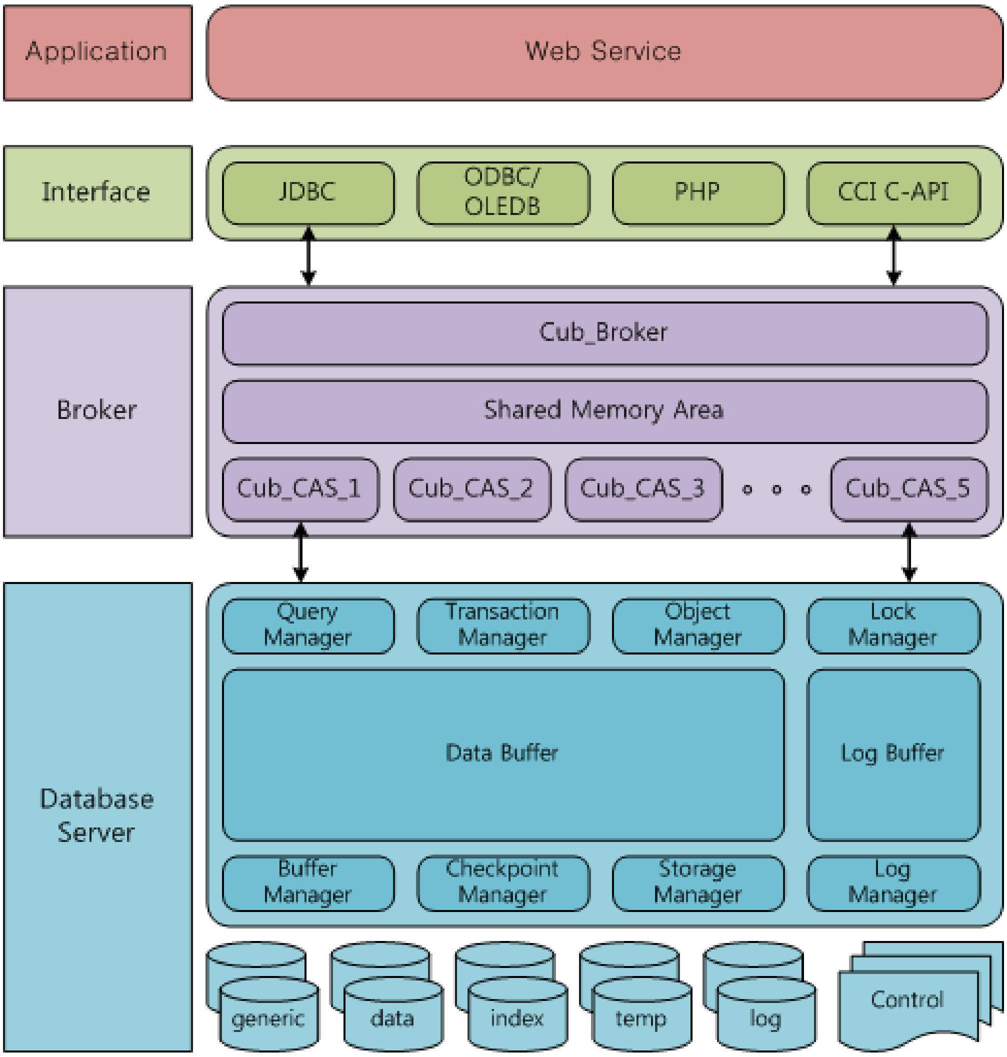
- System Architecture
- Features of CUBRID

System Architecture

System Architecture

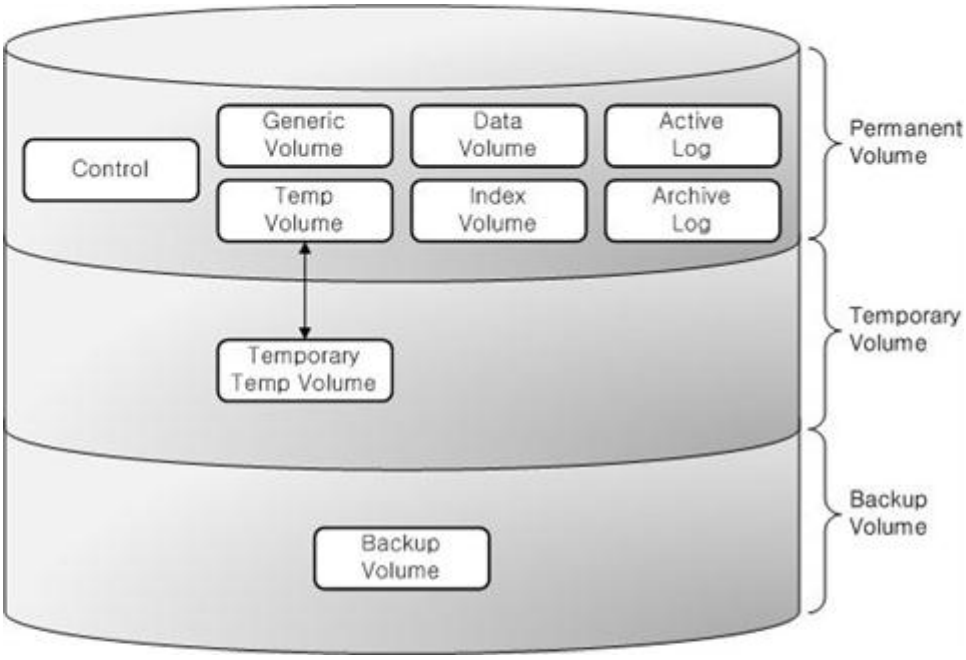
CUBRID is an object-relational database management system (DBMS) consisting of the Database Server, the Broker, and the CUBRID Manager.

- As the core component of the CUBRID Database Management System, the Database Server stores and manages data in multi-threaded client/server architecture. The Database Server processes the queries requested by users and manages objects in the database. The CUBRID Database Server provides seamless transactions using locking and logging methods even when multiple users use the database at the same time. It also supports database backup and restore for the operation.
- The Broker is a CUBRID-specific middleware that relays the communication between the Database Server and external applications. It provides functions including connection pooling, monitoring, and log tracing and analysis.
- The CUBRID Manager is a GUI tool that allows users to remotely manage the database and the Broker. It also provides the Query Editor, a convenient tool that allows users to execute SQL queries on the Database Server. See CUBRID Manager manual or online manual for more information on the CUBRID Manager.



Database Volume Structure

The following diagram illustrates the CUBRID database volume structure. As you can see, the database is divided into three volumes: permanent, temporary and backup. This chapter will examine each volume and its characteristics.



Permanent Volume

Permanent volume is a database volume that exists permanently once it is created. Its types include generic, data, temp, index, control, active log and archive log.

Generic Volume

For efficient management, the volume type to be added to the database can be specified as one of the followings: data, temp or index. If data usage is not specified, it is specified as a generic volume.

Data Volume

Data volume is a volume for storing data such as instances, tables and multimedia data.

Temp Volume

Temporary volume is a volume used temporarily for query processing and sorting. However, the temporary volume is not a volume where the storage is created and destroyed temporarily, but one of the permanent volumes with permanent spaces where the data is stored and destroyed temporarily. Therefore, the data in the temporary volume space gets initialized when CUBRID restarts without leaving any log info.

Index Volume

Index volume is a volume that holds the index information for fast query processing or integrity constraint checks.

Control File

The control file contains the volume, backup and log information in the database.

- **Volume Information** : The information that includes names, locations and internal volume identifiers of all the volumes in the database. When the database restarts, the CUBRID reads the volume information control file. It records a new entry to that file when a new database volume is added.
- **Backup Information** : Locations of all the backups for data, index, and generic volumes are recorded to a backup information control file. This control file is maintained where the log files are managed.

- **Log Information** : This information contains names of all active and archive logs. With the log information control file, you can verify the archive log information. The log information control file is created and managed at the same location as the log files.

Control files include the information about locations of database volumes, backups and logs. Since these files will be read when the database restarts, users must not modify them arbitrarily.

Active Log

Active log is a log that contains recent changes to the database. If a problem occurs, you can use active and archive logs to restore the database completely up to the point of the last commit before the occurrence of the fault.

Archive Log

Archive log is a volume to store logs continuously created after exhausting available active log space that contains recent changes. The archive log volume will be generated only after exhausting available active log volume space, just as the temporary temp volume will be generated after exhausting available permanent temp volume space. However, unlike the temporary temp volume, the archive log volume is not destroyed automatically when the server process terminates. Therefore, a **DBA** needs to manually delete necessary archive logs. The archive log volume can be deleted anytime by **DBA**.

Temporary Volume

Temporary volume has the opposite meaning to the permanent volume. That is, the temporary volume is a storage created only when the accumulated data exceeds the space specified by the user as the permanent volume. The temporary volume is destroyed when the server process terminates. One of such volumes created or destroyed temporarily is the temporary temp volume.

Temporary Temp Volume

Temporary temp volume is a temporary volume created temporarily by the system after exhausting the space specified as the permanent temp volume, whereas the temporary volume belongs to the permanent volume with the permanent space specified. Therefore, the **DBA** should consider the database operations first to free up the permanent temp volume with an appropriate size.

The temporary temp volume is created to free up disk space needed for joining/sorting or index creation. Examples of such large-scale queries of creating temporary volumn are: 1) SQL statements with a **GROUP BY** or **ORDER BY**, 2) SQL statements that contain coordinated subqueries, 3) join queries that perform sort-merge joins, and 4) a **CREATE INDEX** statement.

- **File name of the temporary temp volume** : The file name of the temporary temp volume of CUBRID has the format of *db_name*_**t***num*, where *db_name* is the database name and *num* is the volume identifier. The volume identifier is decremented by 1 from 32766.
- **Configuring the temporary temp volume size** : The number of temporary temp volumes to be created is determined by the system depending on the space size needed for processing transactions. However, users can limit the temporary temp volume size by configuring the **temp_file_max_size_in_pages** parameter value in the system parameter configuration file (**cubrid.conf**). If the **temp_file_max_size_in_pages** parameter value is configured to 0, the temporary temp volume will not be created even after exhausting the permanent temp volume.
- **Configuring save location of the temporary temp volume** : By default, the temporary temp volume is created where the first database volume was created. However, you can specify a different directory to save the temporary temp volume by configuring the **temp_volume_path** parameter value.
- **Deleting the temporary temp volume** : The temporary temp volume exists temporarily only when the database is running. You must not delete the temporary temp volume while the server is running. The temporary temp volume is deleted when the client connection with the server is terminated while the database is running in a standalone mode. On the other hand, the temporary temp volume is deleted when the server process is normally terminated by the **cubrid** utility while the database is running in a client/server mode. If the database server is abnormally terminated, the temporary temp volume will be deleted when the server restarts.

Backup Volume

Backup volume is a database snapshot; based on such backup and log volumes, you can restore transactions to a certain point of time.

You can use the **cubrid backupdb** utility to copy all the data needed for database restore, or configure the **backup_volume_max_size_bytes** parameter value in the database configuration file (**cubrid.conf**) to adjust the backup volume partitioning size.

Database Server

Database Server Process

Each database has a server process. The server process is the core component of the CUBRID Database Server, and handles a user's requests by directly accessing database and log files. The client process connects to the server process via TCP/IP communication. Each server process creates threads to handle requests by multiple client processes. System parameters can be configured for each database, that is, for each server process. The server process can connect to as many client processes as specified by the **max_clients** parameter value.

Master Process

The master process is a broker process that allows the client process to connect to and communicate with the server process. One master process runs for each host. (To be exact, one master process exists for each connection port number specified in the **cubrid.conf** system parameter file.) While the master process listens on the TCP/IP port specified, the client process connects to the master process through that port. The master process changes a socket to server port so that the server process can handle connection.

Execution Mode

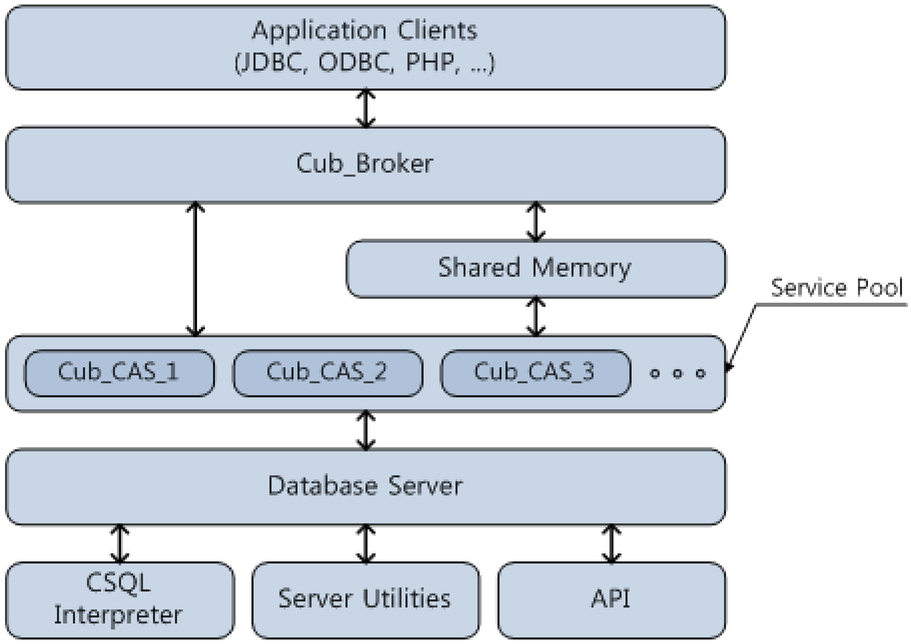
All CUBRID utilities except the server process have two execution modes: client/server mode and standalone mode.

- In client/server mode, the utilities operate as a client process and connect to the server process.
- In the standalone mode, a process is shared between a client and a server, wherein a master process is not required and a database can be directly accessed.

For example, a database creation or a restore utility runs in the standalone mode so it can use the database exclusively by denying the access by multiple users. Another example is that the CSQL Interpreter can either connect to the server process in client/server mode or execute SQL statements by accessing the database in the standalone mode. Note that one database cannot be accessed simultaneously by a server process and a standalone program.

Broker

The Broker is a middleware that allows various application clients to connect to the Database Server. As shown below, the CUBRID system, which includes the Broker, has multi-layered architecture consisting of application clients, **cub_broker**, **cub_cas** and the Database Server.



Application Client

The interfaces that can be used in application clients include C-API, ODBC, JDBC, PHP, Tcl/Tk, Python, and Ruby, OLEDB, and ADO.NET.

cub_cas

cub_cas (CUBRID Common Application Server) acts as a common application server used by all the application clients that request connections. cub_cas also acts as the Database Server's client and provides the connection to the Database Server upon the client's request. The number of cub_cas(s) running in the service pool can be specified in the configuration file, and this number is dynamically adjusted by cub_broker.

cub_cas is a program linked to the CUBRID Database Server's client library and functions as a client module in the server process. In the client module, tasks such as query parsing, optimization, execution plan creation are performed.

cub_broker

cub_broker relays the connection between the application client and the cub_cas. That is, when an application client requests access, the **cub_broker** checks the status of the **cub_cas** through the shared memory, and then delivers the request to an accessible **cub_cas**. It then returns the processing results of the request from the **cub_cas** to the application client.

The **cub_broker** also manages the server load by adjusting the number of **cub_cas**(s) in the service pool and monitors and manages the status of the **cub_cas**. If the **cub_broker** delivers the request to **cub_cas** but the connection to **cub_cas** 1 fails because of an abnormal termination, it sends an error message about the connection failure to the application client and restarts **cub_cas** 1. Restarted **cub_cas** 1 is now in a normal stand-by mode, and will be reconnected by a new request from a new application client.

Shared Memory

The status information of the **cub_cas** is saved in the shared memory, and the **cub_broker** refers to this information to relay the connection to the application client. With the status information saved in the shared memory, the system manager can identify which task the **cub_cas** is currently performing or which application client's request is currently being processed.

Interface Module

CUBRID provides various Application Programming Interfaces (APIs). The following APIs are supported by CUBRID. CUBRID also provides interfaces modules for each interface.

- JDBC : A standard API used to create database applications in Java. CUBRID provides the JDBC driver as an interface module.
- ODBC : A standard API used to create database applications in Windows. CUBRID provides the ODBC driver as an interface module.
- OLE DB : An API used to create COM-based database applications in Windows. CUBRID provides the OLE DB provider as an interface module.
- PHP : CUBIRD provides a PHP interface module to create database applications in the PHP environment. The PHP module is based on the CCI library.
- CCI : CCI is a C language interface provided by CUBRID. The interface module is provided as a C library.

All interface modules access the Database Server through the Broker. The Broker is a middleware that allows various application clients to connect to the Database Server. When it receives a request from an interface module, it calls a native C API provided by the Database Server's client library.

CUBRID Features

Transaction Support

CUBRID supports the following features to completely ensure the atomicity, consistency, isolation and durability in transactions.

- Supporting commit, rollback, savepoint per transaction
- Ensuring transaction consistency in the event of system or database failure
- Ensuring transaction consistency between replications
- Supporting multiple granularity locking of databases, tables and records
- Resolving deadlocks automatically
- Supporting distributed transactions (two-phase commit)

Database Backup and Restore

A database backup is the process of copying CUBRID database volumes, control files and log files; a database restore is the process of restoring the database to a certain point in time using backup files, active logs and archive logs copied by the backup process. For a restore, there must be the same operating system and the same version of CUBRID installed as in the backup environment.
The backup methods which CUBRID supports include online, offline and incremental backups; the restore methods include restore using incremental backups as well as partial and full restore.

Table Partitioning

Partitioning is a method by which a table is divided into multiple independent logical units. Each logical unit is called a partition, and each partition is divided into a different physical space. This will lead performance improvement by only allowing access to the partition when retrieving records. CUBRID provides three partitioning methods:

- Range partitioning: Divides a table based on the range of a column value
- Hash partitioning: Divides a table based on the hash value of a column
- List partitioning: Divides a table based on the column value list

HA Functionalities

High Availability (HA) refers to ability to minimize system down time while continuing normal operation of server in the event of hareware, software, or network failure; that is, the CUBRID HA is functionality that is applied to CUBRID. The CUBRID HA feature has a shared-nothing architecture. The CUBRID performs realtime monitoring for system and CUBRID state with the CUBRID Heartbeat. Then in case of system failure, it automatically performs failover. It follows the two steps below to synchronize data from the master to the slave database servers.

- A transaction log multiplication step where the transaction log created in the database server is replicated in real time to another node
- A transaction log reflection step where data is applied to the slave database server through the analysis of the transaction log being replicated in real time

Replication

Replication is a technique that duplicates data from one database to other databases to improve performance and increase server availability by distributing requests from applications that use the same data into multiple databases. Currently, CUBRID supports replication only on Linux and UNIX. The CUBRID replication system runs based on transaction logs, and it provides real-time replication and ensures transaction consistency/schema independence of the slave database. Additionally, it offers a feature for a master database to be minimally affected by replication. The replication feature consists of the following components:

- Master database: The source database that becomes the target to be replicated. All operations including a read and write operations are performed in this database. Since the replication is performed asynchronously, there will be no effect on the master database administration. Replication logs are created in the master server, which are sent to the slave server via the replication server and the replication agent.

- Slave database: The database replicated from the source database. It allows a client a read operation only in the salve database. If a write operation occurs in the master database, the transaction is automatically replicated to multiple-slave databases, so read operations can be distributed on multiple databases.
- Distribution database: Saves the information about the master and the slave databases. It ensures transaction consistency and effects replication to be distributed.
- Replication server: The replication server runs on the master system and transfers a transaction log in the master database to the replication agent.
- Replication agent: The replication agent is a process that runs on the slave system and performs the actual replication tasks by analyzing and applying the transferred replication log to the slave database server.

Java stored procedure

A stored procedure is a method to decrease the complexity of applications and to improve the reusability, security and performance through the separation of database logic and middleware logic. A stored procedure is written in Java (generic language), and provides Java stored procedures running on the Java Virtual Machine (JVM). To execute Java stored procedures in CUBRID, the following steps should be performed:

- Install and configure the Java Virtual Machine
- Create Java source files
- Compile the files and load Java resources
- Publish the loaded Java classes so they can be called from the database
- Call the Java stored procedures

Click Counter

In the Web, it is a common scenario to count and keep the number of clicks to the database in order to record retrieval history.

The above scenario is generally implemented by using the **SELECT** and **UPDATE** statements; SELECT retrieves the data and UPDATE increases the number of clicks for the retrieved queries.

This approach can cause significant performance degradation due to increased lock contention for **UPDATE** when a number of **SELECT** statements are executed against the same data.

To address this issue, CUBRID introduces the new concept of the click counter that will support optimized features in the Web in terms of usability and performance, and provides the **INCR** function and the **WITH INCREMENT FOR** statement.

Extending the Relational Data Model

Collection

For the relational data model, it is not allowed that a single column has multiple values. In CUBRID, however, you can create a column with several values. For this purpose, collection data types are provided in CUBRID. The collection data type is mainly divided into **SET**, **MULTISET** and **LIST**; the types are distinguished by duplicated availability and order.

- **SET** : A collection type that does not allow the duplication of elements. Elements are stored without duplication after being sorted regardless of their order of entry.
- **MULTISET** : A collection type that allows the duplication of elements. The order of entry is not considered.
- **LIST** : A collection type that allows the duplication of elements. Unlike with **SET** and **MULTISET**, the order of entry is maintained.

Inheritance

Inheritance is a concept to reuse columns and methods of a parent table in those of child tables. CUBRID supports reusability through inheritance. By using inheritance provided by CUBRID, you can create a parent table with some common columns and then create child tables inherited from the parent table with some unique columns added. In this way, you can create a database model which can minimize the number of columns.

Composition

In a relational database, the reference relationship between tables is defined as a foreign key. If the foreign key consists of multiple columns or the size of the key is significantly large, the performance of join operations between tables will be degraded. However, CUBRID allows the direct use of the physical address (OID) where the records of the referred table are located, so you can define the reference relationship between tables without using join operations.

That is, in an object-oriented database, you can create a composition relation where one record has a reference value to another by using the column displayed in the referred table as a domain (type), instead of referring to the primary key column from the referred table.

Getting Started with CUBRID

Getting Started with CUBRID

This chapter contains useful information on starting CUBRID such as how to install and run CUBRID; also it provides instructions on how to use the CSQL Interpreter. This chapter also includes examples on how to write application programs using JDBC, PHP, ODBC and CCI, etc.

This chapter covers the following topics :

- Installing and Running CUBRID
- Configuring Environment Variable and Starting CUBRID
- Using the CSQL Interpreter
- Writing Programs using JDBC
- Writing Programs using PHP
- Writing Programs using ODBC and ASP
- Writing Programs using CCI

Installing and Running

Installing and Running on Linux

Details to Check when Installing

Check the following before installing CUBRID for Linux.

Category	Description
Operating System	Only supports glibc 2.3.4 or later. The glibc version can be checked as follows: rpm -q glibc
64-bit	Since version 2008 R2.0, CUBRID supports both 32-bit and 64-bit Linux. You can check the version as follows: % uname -a Linux host_name 2.6.18-53.1.14.el5xen #1 SMP Wed Mar 5 12:08:17 EST 2008 x86_64 x86_64 x86_64 GNU/Linux Make sure to install the CUBRID 32-bit version on 32-bit Linux and the CUBRID 64-bit version on 64-bit Linux. The followings are the libraries that should be added. Curses Library (rpm -q ncurses) gcrypt Library (rpm -q libgcrypt stdc++ Library (rpm -q libstdc++)

To use CUBRID Manager or Java-stored functions/procedures in CUBRID, you must have JRE (Java Runtime Environment) 1.6 or better installed.

Installing CUBRID

The installation program consists shell scripts that contain binary; thus it can be installed automatically. The following example shows how to install CUBRID with the "CUBRID-8.3.0.0312-linux.x86_64.sh" file on the Linux.

```
[cub_user@cubrid ~]$ sh CUBRID-8.3.1.0168-linux.x86_64.sh
Do you agree to the above license terms? (yes or no) : yes
Do you want to install this software(CUBRID) to the default(/home1/cub_user/CUBRID)
directory? (yes or no) [Default: yes] : yes
Install CUBRID to '/home1/cub_user/CUBRID' ...
In case a different version of the CUBRID product is being used in other machines, please
note that the CUBRID 2008 R3.1 servers are only compatible with the CUBRID 2008 R3.1
clients and vice versa.
Do you want to continue? (yes or no) [Default: yes] : yes
Copying old .cubrid.sh to .cubrid.sh.bak ...

CUBRID has been successfully installed.

demodb has been successfully created.

If you want to use CUBRID, run the following commands
  % . /home1/cub_user/.cubrid.sh
  % cubrid service start
```

As shown in the example above, after installing the downloaded file (CUBRID-8.3.1.0168-linux.x86_64.sh), the CUBRID related environment variables must be set in order to use the CUBRID database. Such setting has been made automatically when logging in the concerned terminal. Therefore there is no need to re-set after the first installation.

```
[cub_user@cubrid ~]$ . /home1/cub_user/.cubrid.sh
```

After the CUBRID Manager is installed, you can start the CUBRID Manager server and Broker as follows:

```
[cub_user@cubrid ~]$ cubrid service start
```

After starting the CUBRID service, if you wish to check whether the service was properly started, then check whether the cub_* processes have been started with grep (as shown below).

```
[cub_user@cubrid ~]$ ps -ef | grep cub_
cub_user 15200 1 0 18:57 ? 00:00:00 cub_master
cub_user 15205 1 0 18:57 pts/17 00:00:00 cub_broker
cub_user 15210 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_1
cub_user 15211 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_2
cub_user 15212 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_3
cub_user 15213 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_4
cub_user 15214 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_5
cub_user 15217 1 0 18:57 pts/17 00:00:00 cub_broker
cub_user 15222 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_1
cub_user 15223 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_2
cub_user 15224 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_3
cub_user 15225 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_4
cub_user 15226 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_5
cub_user 15229 1 0 18:57 ? 00:00:00 cub_auto start
cub_user 15232 1 0 18:57 ? 00:00:00 cub_js start
```

Installing CUBRID (rpm File)

You can install CUBRID by using rpm file that is created on CentOS5. The way of installing and uninstalling CUBRID is the same as that of using general rpm utility. While CUBRID is being installed, a new system group (cubrid) and a user account (cubrid) are created. After installation is complete, you should log in with a cubrid user account to start a CUBRID service.

```
$ rpm -Uvh CUBRID-8.3.1.0168-el5.x86_64.rpm
```

When rmp is executed, CUBRID is installed in the cubrid home directory (/opt/cubrid) and related configuration file (cubrid.[c]sh) is installed in the /etc/profile.d directory. Note that demodb is not automatically installed. Therefore, you must executed /opt/cubrid/demo/make_cubrid_demo.sh. When installation is complete, enter the code below to start CUBRID.

```
[cubrid@cubrid ~]$ cubrid service start
```

Note You must check RPM dependency when installing with RPM. If you ignore (--nodeps) dependency, it may not be executed.

Note Even if you remove RPM, user accounts and databases that are created after installing, you must remove it manually, if needed.

CUBRID Upgrade

When you specify an installation directory where the previous version of CUBRID is already installed, a message which asks to overwrite files in the directory will appear. Entering **no** will stop the installation.

```
Directory '/home1/cub_user/CUBRID' exist!
If a CUBRID service is running on this directory, it may be terminated abnormally.
And if you don't have right access permission on this directory(subdirectories or files),
install operation will be failed.
Overwrite anyway? (yes or no) [Default: no] : yes
```

Choose whether to overwrite the existing configuration files during the CUBRID installation. Entering **yes** will overwrite and back up them as extension .bak files.

```
The configuration file (.conf or .pass) already exists. Do you want to overwrite it? (yes
or no) : yes
```

Environment Configuration

To modify the environment such as service ports etc, edit the parameters of a configuration file located in the **$CUBRID/conf** directory. See Environment Configuration for more information.

Note You must check the dependency when you attempt to install using RPM. Installation may not succeed if the dependency is ignored (--nodeps).

Installing and Running on Windows

Details to Check when Install

CUBRID 2008 R2.0 supports both 32-bit and 64-bit Windows. You can check the version by selecting [My Computer] > [System Properties]. Make sure to install the CUBRID 32-bit version on 32-bit Windows and the CUBRID 64-bit version on 64-bit Windows.

The CUBRID Manager and Java stored procedures require the Java Runtime Environment (JRE) version 1.6 or later.

Setup Type

- **Server and Driver Installation** : CUBRID Server, CSQL (a command line tool), interface drivers (OLEDB Provider, ODBC, JDBC, C API) are all installed.
- **Driver Installation** : The interface drivers (OLEDB Provider, ODBC, JDBC, C API) are only installed. You can select this type of installation if development or operation is performed by remote connection to the computer in which the CUBRID database server is installed.

CUBRID Upgrade

To install a new version of CUBRID in an environment in which a previous version has already been installed, select [CUBRID Service Tray] > [Exit] from the menu to stop currently running services, and then remove the previous version of CUBRID. Note that when you are prompted with "Do you want to delete all the existing version of databases and the configuration files?" you must select "No" to protect the existing databases.

For more information on migrating a database from a previous version to a new version, see Migrating Database.

Environment Configuration

To change configuration such as service ports to meet the user environment, the parameter values of the files stated below should be changed in the **%CUBRID%\conf** directory.

File	Description
cm.conf	CUBRID Manager's configuration file; the port number 8001 is configured by default. Two port numbers are required to use CUBRID; a configured number and the number added by 1 are used. For example, 8001 is configured for connection, the port number 8001 and 8002 are reserved.
cubrid.conf	Server configuration file is used to set the following: database memory, the number of threads due to the number of concurrent users, connection port between the Broker and Server, etc. See cubrid_broker.conf Configuration File and Default Parameters for details.
cubrid_broker.conf	Broker configuration file; the port is used by the broker that is operated. The file is used to set the number of CAS, SQL LOGs, etc. The ports shown in drivers such as JDBCs are the concerned Broker's ports. See Parameter by Broker for details.

Configuring Environment Variable and Starting CUBRID

Configuring the Environment Variable

The following environment variables need to be set in order to use the CUBRID. The necessary environment variables are automatically set when the CUBRID system is installed or can be changed, as needed, by the user.

CUBRID Environment Variables

- CUBRID : The default environment variable that designates the location where the CUBRID is installed. This variable must be set accurately since all programs included in the CUBRID system uses this environment variable as reference.
- CUBRID_DATABASES : The environment variable that designates the location of the database location information file. The CUBRID system saves and manages the absolute path of database volumes that are used in the **$CUBRID_DATABASES/databases.txt** file. See databases.txt file.
- CUBRID_LANG : The environment variable that designates the language that will be used in the CUBRID system. Currently, CUBRID provides English (en_US) and Korean (ko_KR.euckr and ko_KR.utf8). it is not a mandatory setting. Therefore, if the variable has not been set, then refer to the LANG environment variable or use en_US, which is the default value. See Language Setting.

The above mentioned environment variables are set when the CUBRID is installed. However, the following commands can be used to verify the setting.

For Linux :

```
% printenv CUBRID
% printenv CUBRID_DATABASES
% printenv CUBRID_LANG
```

In Windows :

```
C:\> set CUBRID
```

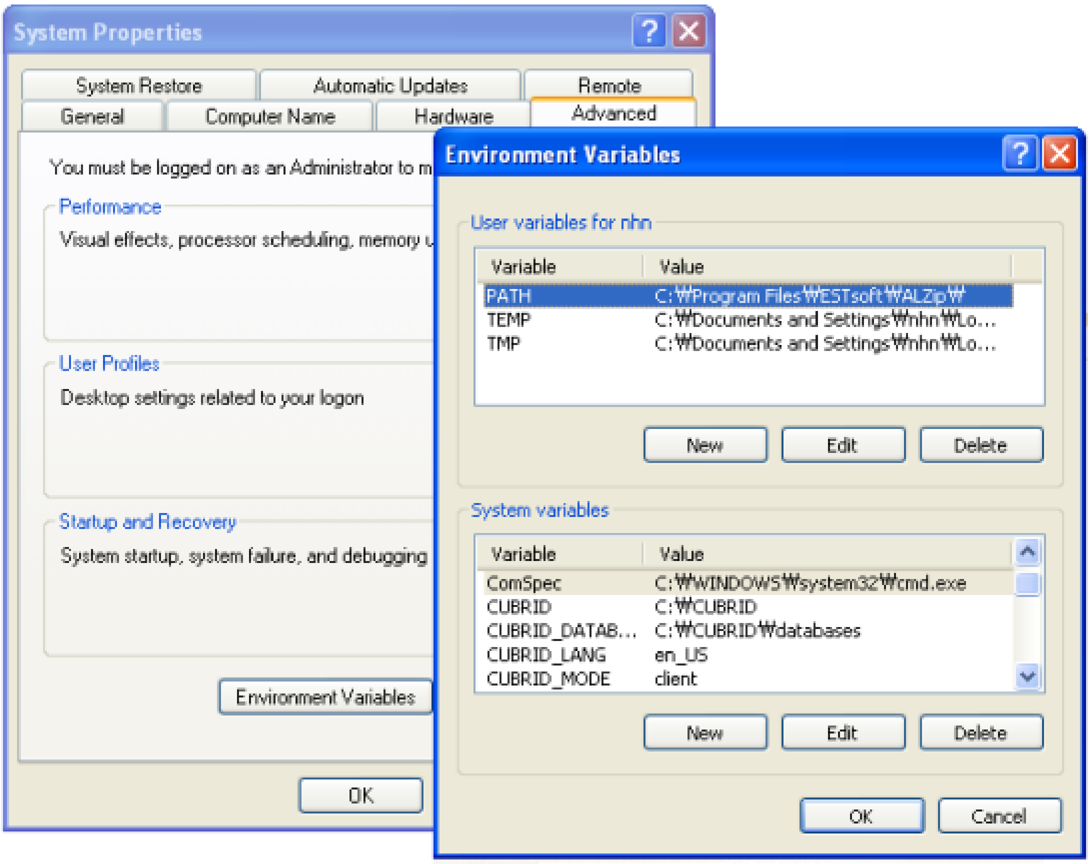
OS Environment and Java Environment Variables

- PATH : In the Linux environment, the directory **$CUBRID/bin**, which includes a CUBRID system executable file, must be included in the PATH environment variable.
- LD_LIBRARY_PATH : In the Linux environment, **$CUBRID/lib**, which is the CUBRID system's dynamic library file (libjvm.so), must be included in the **LD_LIBRARY_PATH** (or **SHLIB_PATH** or **LIBPATH**) environment variable.
- Path : In the Windows environment, the **$CUBRID/bin**, which is a directory that contains CUBRID system's execution file, must be included in the **Path** environment variable.
- JAVA_HOME : To use the Java stored procedure in the CUBRID system, the Java Virtual Machine (JVM) version 1.6 or later must be installed, and the **JAVA_HOME** environment variable must designate the concerned directory. See the Environment Configuration for Java Stored Functions/Procedures.

Configuring the Environment Variable

For Windows

If the CUBRID system has been installed in the Windows environment, then the installation program automatically sets the necessary environment variable. Select [Systems Properties] in [My Computer] and select the [Advanced] tab. Click the [Environment Variable] button and check the setting in the [System Variable]. The settings can be changed by clicking on the [Edit] button. See the Windows help for more information on how to change the environment variable in the Windows environment.



For Linux

If the CUBRID system has been installed in the Linux environment, the installation program automatically creates the **.cubrid.sh** or **.cubrid.csh** file and makes configurations so that the files are automatically called from the installation account's shell log-in script. The following is the .cubrid.sh environment variable setting file that was created in an environment that uses **sh**, **bash**, etc.

```
CUBRID=/home1/cub_user/CUBRID
CUBRID_DATABASES=/home1/cub_user/CUBRID/databases
CUBRID_LANG=en_US
ld_lib_path=`printenv LD_LIBRARY_PATH`
if [ "$ld_lib_path" = "" ]
then
LD_LIBRARY_PATH=$CUBRID/lib
else
LD_LIBRARY_PATH=$CUBRID/lib:$LD_LIBRARY_PATH
fi
SHLIB_PATH=$LD_LIBRARY_PATH
LIBPATH=$LD_LIBRARY_PATH
PATH=$CUBRID/bin:$CUBRID/cubridmanager/:$PATH
export CUBRID
export CUBRID_DATABASES
export CUBRID_LANG
export LD_LIBRARY_PATH
export SHLIB_PATH
export LIBPATH
export PATH
```

Language Setting

The language that will be used in the CUBRID DBMS can be designated with the **CUBRID_LANG** environment variable. The following are values that can currently be set in the **CUBRID_LANG** environment variable.

- **en_US** : English (Default value)
- **ko_KR.euckr** : Korean EUC-KR encoding
- **ko_KR.utf8** : Korean utf-8 encoding

The language setting in the CUBRID system does not represent the character sets of data that is saved. In other words, even though the **CUBRID_LANG** is set to ko_KR.utf8, the data may not be changed to the concerned encoding. CUBRID's language setting will have an influence on the message printed from the program and will impact the date/time data type constant displayed throughout the use of the program.

If the **CUBRID_LANG** is not set, then the value of the LANG environment variable will be used. If the set value does not support the **CUBRID_LANG** or **LANG** value, then the action will be made as if the setting has been made to en_US, the default value.

Starting the CUBRID Service

Configure environment variables and language, and then start the CUBRID service. For more information on configuring environment variables and language, see Registering Services or Starting and Stopping Services.

Shell Command

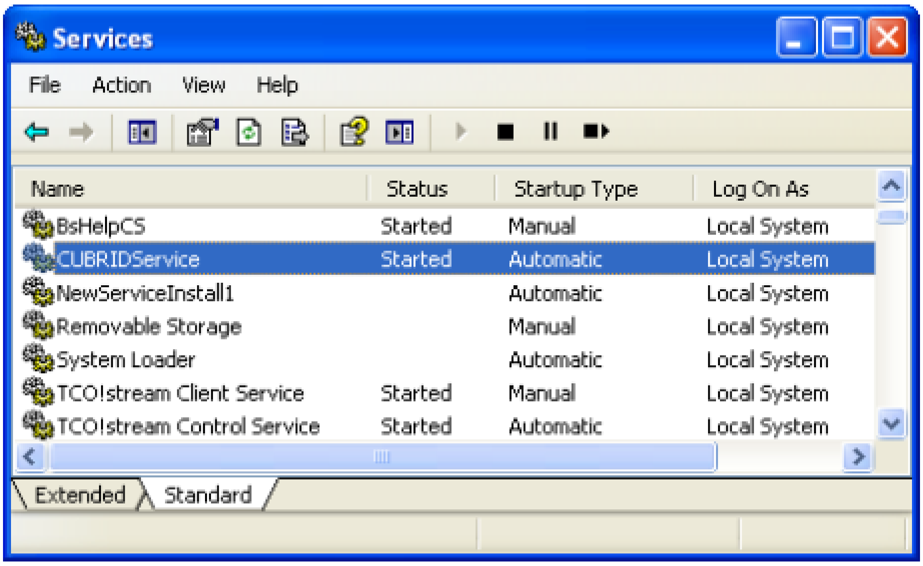
The following shell command can be used to start the CUBRID service and the demodb included in the installation package.

```
% cubrid service start

@ cubrid master start
++ cubrid master start: success
@ cubrid broker start
++ cubrid broker start: success
@ cubrid manager server start
++ cubrid manager server start: success

% cubrid server start demodb
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R4.0

++ cubrid server start: success

@ cubrid server status
Server demodb (rel 8.3, pid 31322)
```

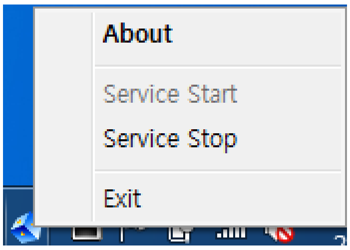
CUBRIDService or CUBRID Service Tray

On the Windows environment, you can start or stop a service as follows:

- Go to [Control Panel] > [Performance and Maintenance] > [Administrator Tools] > [Services] and select the CUBRIDService to start or stop the service.



- In the system tray, right-click the CUBRID Service Tray. To start CUBRID, select [Service Start]; to stop it, select [Service Stop]. If you click [Exit] while CUBRID is running, all the services and process in the server are stopped.



Note An administrator level (SYSTEM) authorization is required to start/stop CUBRID processes through the CUBRID Service tray; a login level user authorization is required to start/stop them with shell commands. If you cannot control the CUBRID processes on the Windows Vista or later version environment, select [Execute as an administrator (A)] in the [Start] > [All Programs] > [Accessories] > Command Prompt]) or execute it by using the CUBRID Service Tray. When all processes of CUBRID Server stops, an icon on the CUBRID Service tray turns out red.

CSQL Interpreter

Starting the CSQL Interpreter

The CSQL Interpreter is a program used in CUBRID. The entered SQL statements and results can be saved in the file for later use. For more information Introduction to the CSQL Interpreter and CSQL Execution Mode.

CUBRID offers the "CUBRID Manager" program, a convenient GUI program. All SQL can be executed and the results can be viewed from the CUBRID Manager's query editor. For more information, see CUBRID Manager manual or online manual.

In this section, we will provide information on using the CSQL Interpreter in the Linux environment.

Starting the CSQL Interpreter

The CSQL program can be started in the shell as shown below.

```
% csql demodb
          CUBRID SQL Interpreter
Type ';help' for help messages.
csql> ;help
=== <Help: Session Command Summary> ===
    All session commands should be prefixed by ';' and only blanks/tabs
    can precede the prefix. Capitalized characters represent the minimum
    abbreviation that should be entered to execute the specified command.

    ;REAd   [<file-name>]       - read a file into command buffer.
    ;Write  [<file-name>]       - (over)write command buffer into a file.
    ;APpend [<file-name>]       - append command buffer into a file.
    ;PRINT                      - print command buffer.
    ;SHELL                      - invoke shell.
    ;CD                         - change current working directory.
    ;EXit                       - exit program.

    ;CLear                      - clear command buffer.
    ;EDIT                       - invoke system editor with command buffer.
    ;List                       - display the content of command buffer.

    ;RUn                        - execute sql in command buffer.
    ;Xrun                       - execute sql in command buffer,
                                  and clears the command buffer.
    ;COmmit                     - commit the current transaction.
    ;ROllback                   - roll back the current transaction.
    ;AUtocommit [ON|OFF]        - enable/disable auto commit mode.
    ;REStart                    - restart database.

    ;SHELL_Cmd [shell-cmd]      - set default shell, editor, print and pager
    ;EDITOR_Cmd [editor-cmd]      command to new one, or display the current
    ;PRINT_Cmd [print-cmd]        one, respectively.

    ;DATE                       - display the local time, date.
    ;DATAbase                   - display the name of database being accessed.
    ;SChema class-name          - display schema information of a class.
    ;SYntax [sql-cmd-name]      - display syntax of a command.
    ;TRigger [`*'|trigger-name] - display trigger definition.
    ;Get system_parameter       - get the value of a system parameter.
    ;SEt system_parameter=value - set the value of a system parameter.
    ;PLan [simple|detail|off]   - show query execution plan.
    ;Info <command>             - display internal information.
    ;TIme [ON/OFF]              - enable/disable to display the query
                                  execution time.
    ;HISTORYList                - display list of the executed queries.
    ;HISTORYRead <history_num>  - read entry on the history number into command buffer.
    ;HElp                       - display this help message.
csql>
```

Executing the SQL with CSQL

After the CSQL has been executed, you can enter the SQL into the CSQL prompt. Each SQL statement must end with a semicolon (;). Multiple SQL statements can be entered in a single line. To execute the SQL statements entered, use the ;x session command. You can find the simple usage of the session commands with the ;help command. For more information, see Session Commands.

```
% csql demodb
CUBRID SQL Interpreter
Type `;help' for help messages.
csql> select * from olympic;
csql> ;x
=== <Result of SELECT Command in Line 1> ===

    host_year  host_nation           host_city             opening_date  closing
_date  mascot                slogan                introduction
================================================================================
=======================================================================
         2004  'Greece'              'Athens'              08/13/2004    08/29/2
004    'Athena  Phevos'      'Welcome Home'        'In 2004 the Olympic Games re
turned to Greece, the home of both the ancient Olympics and the first modern Olympics.

<omitted>
25 rows selected.

Current transaction has been committed.

1 command(s) successfully processed.
csql> SELECT SUM(n) FROM (SELECT gold FROM participant WHERE nation_code='KOR'
csql> UNION ALL SELECT silver FROM participant WHERE nation_code='JPN') AS t(n);
csql> ;x

=== <Result of SELECT Command in Line 1> ===

       sum(n)
=============
           82


1 rows selected.

Current transaction has been committed.

1 command(s) successfully processed.
csql> ;exit
```

Programming with JDBC

Setting up the JDBC Environment

System Requirements

- JDK 1.6 or later
- CUBRID 2008 R1.0 or later
- CUBRID JDBC Driver 2008 R1.0 or later

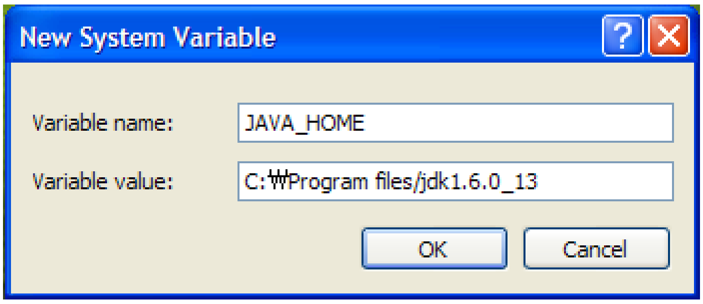
Installing and Configuring Java Environment

You must already have Java installed and the JAVA_HOME environment variable set on your system. To install Java, download it from the Java homepage (http://java.sun.com). For more information, see Environment Settings for Java Stored Functions/Procedures.

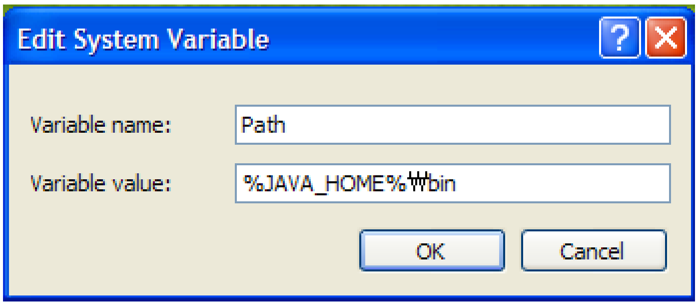
Configuring Envrionment Variables for Windows

After installing JAVA, double click [My Computer] and click [System Properties]. In the [Advanced] tab, click [Envrionment Variables]. The [Environment Variables] dialog will appear.

In the [System Variables], click [New]. Enter **JAVA_HOME** and Java installation path such as C:\Program Files\Java\jdk1.6.0_16 and then press [Enter].



Select "Path" and then click [Edit]. Add **%JAVA_HOME%\bin** to the variable and then click [OK].



You can configure **JAVA_HOME** and **PATH** in the shell.

```
set JAVA_HOME= C:\Program Files\Java\jdk1.6.0_16
set PATH=%PATH%;%JAVA_HOME%\bin
```

Configuring the Environment Variables for Linux

Specify the directory path where Java is installed (example : /usr/java/jdk1.6.0_16) in the **JAVA_HOME** environment variable, and add **$JAVA_HOME/bin** to the **PATH** environment variable.

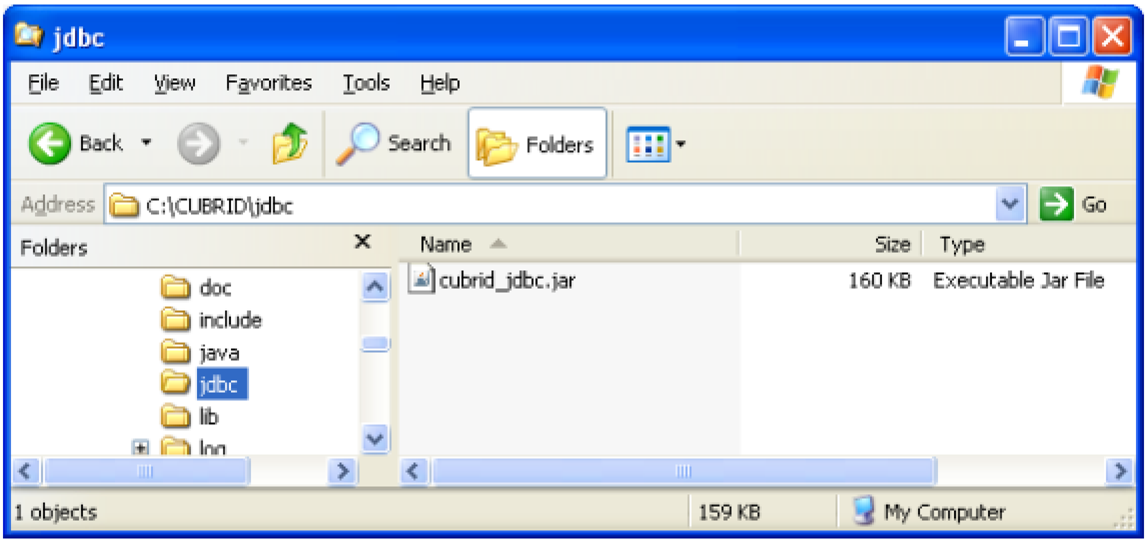
```
export JAVA_HOME=/usr/java/jdk1.6.0_16      //bash
export PATH=$JAVA_HOME/bin:$PATH            //bash

setenv JAVA_HOME /usr/java/jdk1.6.0_16      //csh
```

```
set path = ($JAVA_HOME/bin $path)            //csh
```

JDBC Driver Setting

To use the JDBC, set your **CLASSPATH** environment variable to the path where the CUBRID JDBC driver is located.

The CUBRID JDBC driver (**cubrid_jdbc.jar**) is located in jdbc directory which is subdirectory where CUBRID is installed.



Configuring the CLASSPATH Environment Variables for Windows

```
set CLASSPATH=%CUBRID%\jdbc\cubrid_jdbc.jar:.
```

Configuring the CLASSPATH Environment Variables for Linux

```
export CLASSPATH=$HOME/CUBRID/jdbc/cubrid_jdbc.jar:.
```

Note If a CUBRID JDBC driver has been installed in the same library directory (**$JAVA_HOME/jre/lib/ext**) where the JRE is located, it may be loaded ahead of the server-side JDBC driver used by the Java stored procedure, causing it to malfunction. In a Java stored procedure environment, make sure not to install the generic CUBRID JDBC driver in the directory where the JRE is installed (**$JAVA_HOME/jre/lib/ext**).

JDBC Sample

The following is a simple example that connects to CUBRID by using the JDBC driver and retrieves and inserts data. To run the sample program, make sure that the database you are trying to connect to and the CUBRID Broker are running. In the sample, you will use the **demodb** database that is created automatically during the installation.

JDBC Driver Load

To connect to CUBRID, load the JDBC driver using the for Name() method provided in the class. For more information, see the CUBRID JDBC Driver.

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
```

How to Make the Connection to Database

When the JDBC driver is loaded, use the getConnection() method provided in the DriverManager to connect to the database. To create a Connection object, you must specify the url for describing the location of the database, database user name, password, etc. For more information, see the Connection Configuration.

```
String url = "jdbc:cubrid:localhost:30000:demodb:::";
String userid = "dba";
String password = "";
```

```
Connection conn = DriverManager.getConnection(url,userid,password);
```

Manipulating database (executing queries and processing the ResultSet)

To send a query statement to the connected database and execute it, create the **Statement**, **PrepardStatement**, and **CallableStatement** objects. When a statement object has been created, execute the query using the **executeQuery()** method or the **executeUpdate()** method for the statement object. The **next()** method can process the following row from the ResultSet that is returned from the **executeQuery()** method. For more information, see the BRID JDBC Driver.

Note If you execute commit after query execution, **ResultSet** is automatically closed. Therefore, you must not use **ResultSet** after commit. CUBRID is, in general, executed in auto-commit mode. If you does not want auto-commit mode, you must state **conn.setAutocommit(false);** in the code.

Disconnecting from the database

Each method can be disconnected from the database by executing the **close()** method.

JDBC Sample 1

The sample code shown below creates a table, executes a query with a prepared statement, and then rolls back the query. Modify the parameter value of the **getConnection()** method for practice.

```
import java.util.*;
import java.sql.*;

public class Basic {
   public static Connection connect() {
      Connection conn = null;
      try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
           conn = DriverManager.getConnection("jdbc:cubrid:localhost:30000:::","dba","");
           conn.setAutoCommit (false) ;
      } catch ( Exception e ) {
           System.err.println("SQLException : " + e.getMessage());
      }
      return conn;
   }

   public static void printdata(ResultSet rs) {
      try {
          ResultSetMetaData rsmd = null;

          rsmd = rs.getMetaData();
          int numberofColumn = rsmd.getColumnCount();

          while (rs.next ()) {
              for(int j=1; j<=numberofColumn; j++ )
                  System.out.print(rs.getString(j) + "  " );
              System.out.println("");
          }
      } catch ( Exception e ) {
           System.err.println("SQLException : " + e.getMessage());
      }
   }

   public static void main(String[] args) throws Exception {
      Connection conn = null;
      Statement stmt = null;
      ResultSet rs = null;
      PreparedStatement preStmt = null;

      try {
           conn = connect();

           stmt = conn.createStatement();
           stmt.executeUpdate("create class xoo ( a int, b int, c char(10))");

           preStmt = conn.prepareStatement("insert into xoo values(?,?,''''100'''')");
```

```
            preStmt.setInt (1, 1) ;
            preStmt.setInt (2, 1*10) ;
            int rst = preStmt.executeUpdate () ;

            rs = stmt.executeQuery("select a,b,c from xoo" );

            printdata(rs);

            conn.rollback();
            stmt.close();
            conn.close();
       } catch ( Exception e ) {
            conn.rollback();
            System.err.println("SQLException : " + e.getMessage());
       } finally {
            if ( conn != null ) conn.close();
       }
   }
}
```

JDBC Sample 2

The following is an example of executing **SELECT** statement by connecting to demodb that is provided by CUBRID during installation.

```
import java.sql.*;
public class SelectData {
   public static void main(String[] args) throws Exception {
       Connection conn = null;
       Statement stmt = null;
       ResultSet rs = null;
       try {
       // Connect to CUBRID
       Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       conn =
DriverManager.getConnection("jdbc:CUBRID:localhost:30000:demodb:::","dba","");
       String sql = "select name, players from event";
       stmt = conn.createStatement();
       rs = stmt.executeQuery(sql);
       while(rs.next()) {
           String name = rs.getString("name");
           String players = rs.getString("players");
           System.out.println("name ==> " + name);
           System.out.println("Number of players==> " + players);
           System.out.println("\n=========================================\n");
       }
       rs.close();
       stmt.close();
       conn.close();
       } catch ( SQLException e ) {
           System.err.println(e.getMessage());
       } catch ( Exception e ) {
           System.err.println(e.getMessage());
       } finally {
           if ( conn != null ) conn.close();
       }
   }
}
```

JDBC Example 3

The following is an example of executing **INSERT** statement by connecting to demodb that is provided by CUBRID during installation. You can delete or modify data the same way as you insert data. This means that you can reuse the code below by simply changing the query statements.

```
import java.sql.*;
public class insertData {
   public static void main(String[] args) throws Exception {
       Connection conn = null;
       Statement stmt = null;
       try {
           // CUBRID Connect
```

```
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
           conn =
DriverManager.getConnection("jdbc:cubrid:localhost:30000:demodb:::","dba","");
           String sql = "insert into olympic(host_year, host_nation, host_city,
opening_date,            closing_date) values (2008, 'China', 'Beijing', to_date('08-08-
2008','mm-dd-yyyy'),            to_date('08-24-2008','mm-dd-yyyy'))";
           stmt = conn.createStatement();
           stmt.executeUpdate(sql);
           System.out.println("Data is inserted.");
           stmt.close();
       } catch ( SQLException e ) {
           System.err.println(e.getMessage());
       } catch ( Exception e ) {
           System.err.println(e.getMessage());
       } finally {
           if ( conn != null ) conn.close();
       }
   }
}
```

Programming with PHP

Installing the PHP Module

Go to the CUBRID website and the see how to install the PHP module.

Installing PHP for Windows

After compiling and building cubrid_php_[version].dll from php_cubrid.sln in the win directory, create a directory named CUBRID in the directory where PHP is installed, and then copy the the cubrid_php_[version].dll file. For more information, refer to the INSTALL file
Add required settings as shown in the example below by editing the php.ini file.

```
extension_dir=C:\PHP\CUBRID
extension=cubrid_php5.1.4.dll
```

Once the configuration is complete, restart the web server. If you create test.php using using the phpinfo() function of PHP and enter a url as http://localhost/test.php on your Web browser, you will see the CUBRID information if the installation was successful.

Installing PHP for Linux

After compiling and building cubrid.so file by running phpize in the src directory, create a directory named php/extensions in the directory where PHP is installed, and then copy the the cubrid.so file from the module directory. For more information, refer to the INSTALL file
Add required settings as shown in the example below by editing the php.ini file.

```
extension_dir=/usr/lib/php5/lib/php/extentions
extension=cubrid.so
```

After restarting Web server, check the configuration using phpinfo() function.

As with the Windows version of PHP, if you can see the CUBRID information on the web browser, it means that the installation was successful.

PHP Sample

The following is a simple example that establishes a connection between PHP and CUBRID. This section will cover the most basic and notable features. Before running the sample program, a database and the Broker you are trying to connect must be running. This example uses the **demodb** database created during the installation.

Example of Data Retrieval

```
<html>
<head>
<meta http-equiv='content-type' content='text/html; charset=euc-kr'
</head>
<body>
<center>
<table border=2>
<?
   // Set server information for CUBRID connection. host_ip is the IP address where the
CUBRID Broker is installed (localhost in this example), and host_port is the port number
of the CUBRID Broker. The port number is the default given during the installation. For
details, see "Administrator's Guide."
   $host_ip = "localhost";
   $host_port = 30000;
   $db_name = "demodb";
   // Connect to CUBRID Server. Do not make the actual connection, but only retain the
connection information. The reason for not making the actual connection is to handle
transaction more efficiently in the 3-tier architecture.
   $cubrid_con = @cubrid_connect($host_ip, $host_port, $db_name);
   if (!$cubrid_con) {
```

```
        echo "Database Connection Error";
        exit;
   }
?>
<?
   $sql = "select sports, count(players) as players from event group by sports";
   // Request the CUBRID Server for the results of the SQL statement. Now make the actual
connection to the CUBRID Server.
   $result = cubrid_execute($cubrid_con, $sql);
   if ($result) {
       // Get the column names from the result set created by the SQL query.
       $columns = cubrid_column_names($result);
       // Get the number of columns in the result set created by the SQL query.
       $num_fields = cubrid_num_cols($result);
       // List the column names of the result set on the screen.
       echo("<tr>");
       while (list($key, $colname) = each($columns)) {
       echo("<td align=center>$colname</td>");
       }
       echo("</tr>");
       // Get the results from the result set.
       while ($row = cubrid_fetch($result)) {
           echo("<tr>");
           for ($i = 0; $i < $num_fields; $i++) {
               echo("<td align=center>");
               echo($row[$i]);
               echo("</td>");
           }
           echo("</tr>");
       }
   }
   // The PHP module in the CUBRID runs in a 3-tier architecture. Even when calling SELECT
for transaction processing, it is processed as a part of the transaction. Therefore, the
transaction needs to be rolled back by calling commit or rollback even though SELECT was
called for smooth performance.
   cubrid_commit($cubrid_con);
   cubrid_disconnect($cubrid_con);
?>
</body></html>
```

Example of Data Insertion

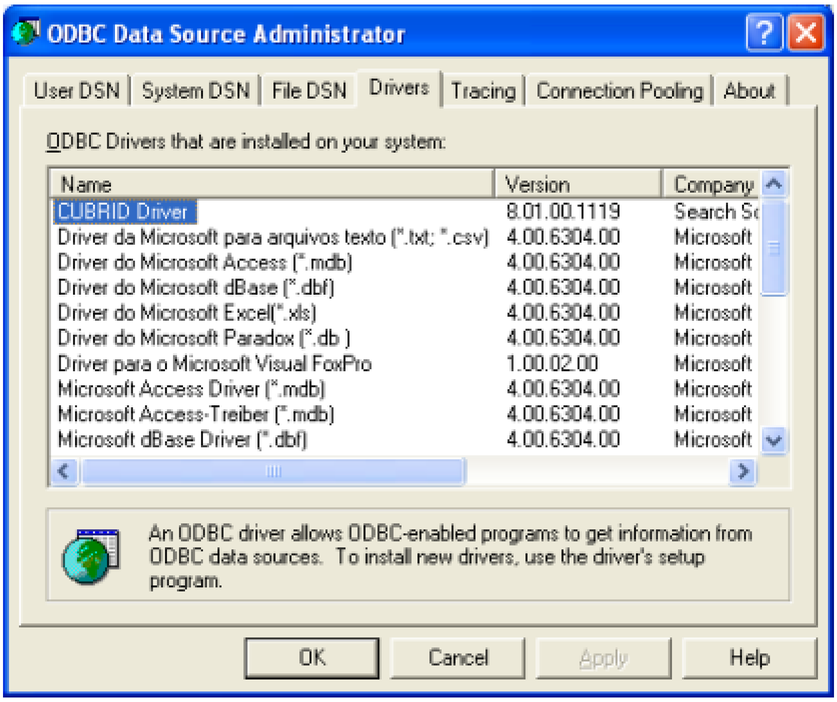
```
<html>
<head>
<meta http-equiv='content-type' content='text/html; charset=euc- kr'>
</head>
<body>
<center>
<table border=2>
<?
   $host_ip = "localhost";
   $host_port = 30000;
    $db_name = "demodb";
   $cubrid_con = @cubrid_connect($host_ip, $host_port, $db_name);
   if (!$cubrid_con) {
        echo "Database Connection Error";
       exit;
   }
?>
<?
   $sql = "insert into olympic (host_year,host_nation,host_city,opening_date,closing_date)
values     (2008, 'China', 'Beijing', to_date('08-08-2008','mm-dd- yyyy'),to_date('08-24-
2008','mm-dd-yyyy'))    ;"
   $result = cubrid_execute($cubrid_con, $sql);
   if ($result) {
       // Handled successfully, so commit.
       cubrid_commit($cubrid_con);
       echo("Inserted successfully ");
   } else {
       // Error occurred, so the error message is output and rollback is called.
       echo(cubrid_error_msg());
       cubrid_commit($cubrid_con);
   }
```

```
    cubrid_disconnect($cubrid_con);
?>
</body></html>
```

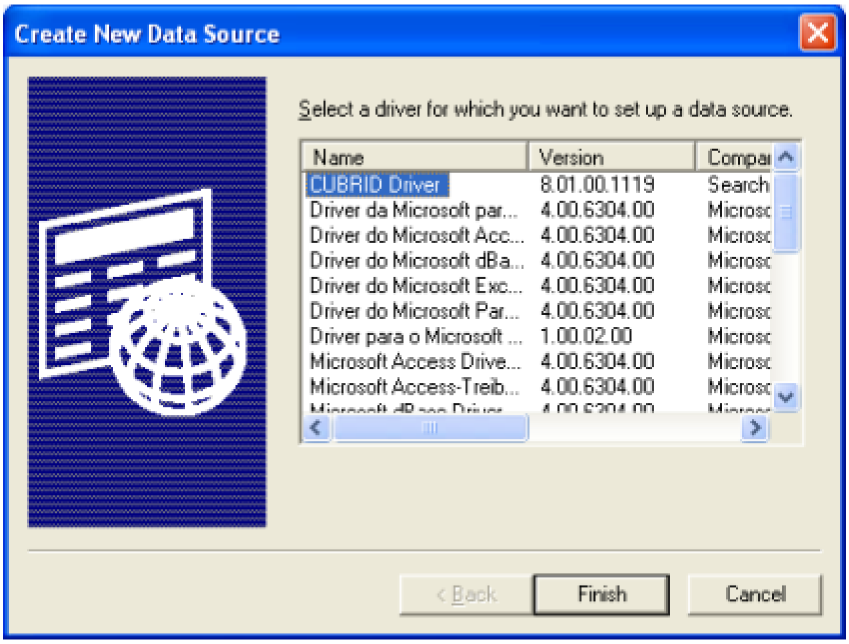
Programming with ODBC and ASP

Configuring the Environment of ODBC and ASP

CUBRID ODBC is compatible for version 3.52 ODBC and LEVEL2. Note that backward compatibility is not guaranteed for applications that are written with ODBC Spec 2.x. The CUBRID ODBC driver is automatically installed while CUBRID is installed. You can verify it from [Control Panel] > [Administrative Tools] > [Data Source (ODBC)] > [Drivers] tab.

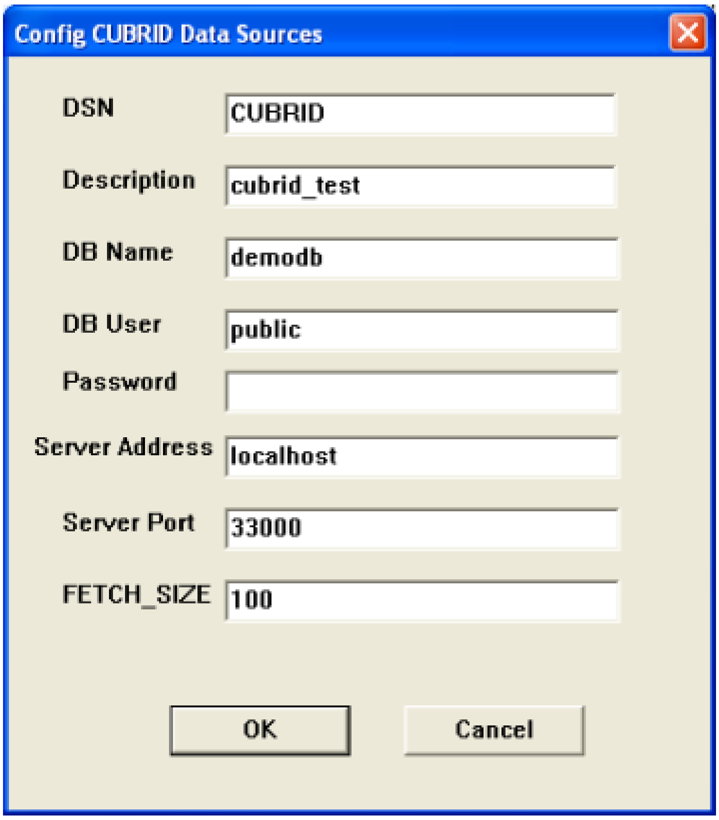


If the CUBRID ODBC driver is detected, set a DSN as a database where the application is trying to connect. To set up a DSN, click the [Add] button in the ODBC Data Source Administrator dialog box. Then, the following dialog box appears. Select "CUBRID Driver," and then click the [Finish] button.



When the following [Config CUBRID Data Sources] dialog box appears, enter the database name that you try to connect to in the [DB Name] field, the port number of the CUBRID Broker in the [Server Port] field, and then click [OK] button. You can verify the number in the **cubrid.broker.conf** file.

FETCH_SIZE refers to the number of records fetched from server whenever **cci_fetch**() function of CCI library is called; the CCI library is internally used by ODBC driver.



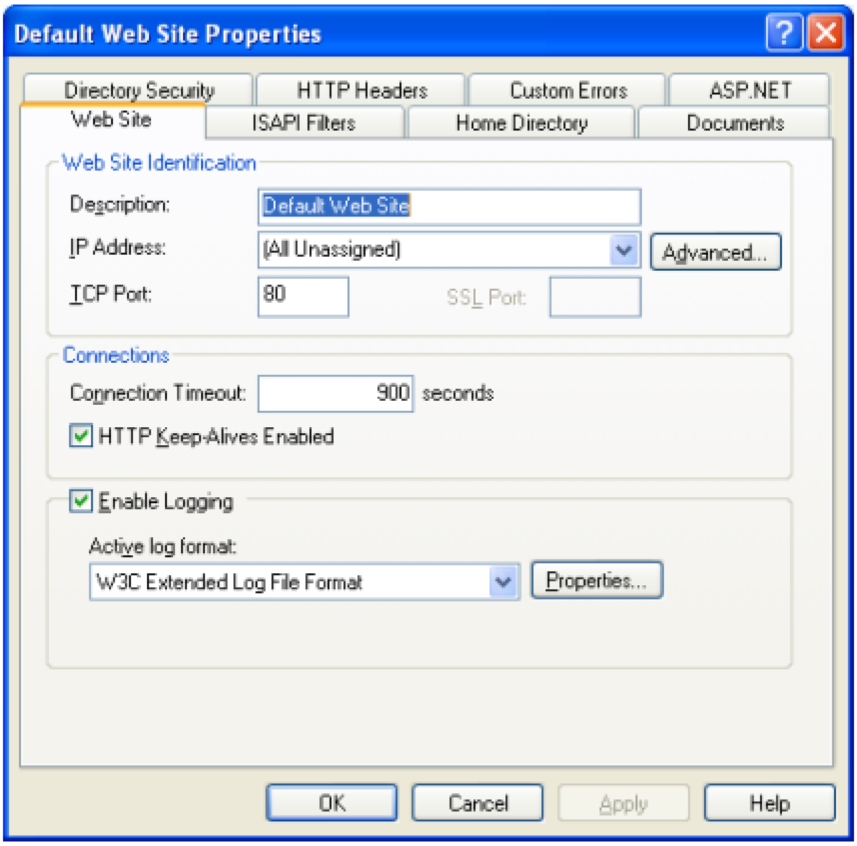
For more information on CUBRID ODBC driver, see "ODBC API Reference."

- CUBRID ODBC Driver

- Using OIDs and Collections
- Supported Functions and Backward Compatibility

ASP Sample

In the virtual directory where the ASP sample program runs, right-click "Default Web Site" and click [Properties].



The dialog box shown above will appear. Under the **Web Site Identification**, in the **IP Address** drop-down box, select "(All Unassigned)." This sets the IP address to localhost. If you want to run the sample program using a specific IP address, configure the directory with the IP address as a virtual directory and register the IP address in Properties.

The following is an example in which the IP address is set to localhost.

Example

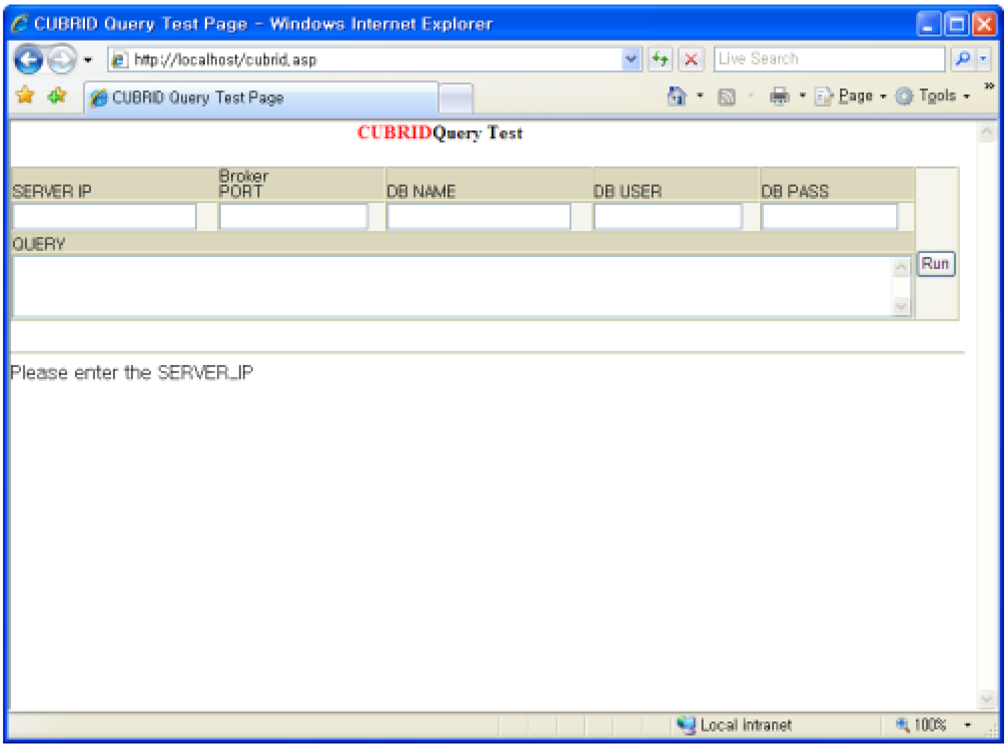
Save the following sample code as cubrid.asp in the virtual directory.

```
<HTML>
    <HEAD>
     <meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
       <title>CUBRID Query Test Page</title>
  </HEAD>
 <BODY topmargin="0" leftmargin="0">

 <table border="0" width="748" cellspacing="0" cellpadding="0">
   <tr>
     <td width="200"></td>
     <td width="287">
       <p align="center"><font size="3" face="Times New Roman"><b><font
color="#FF0000">CUBRID</font>Query Test</b></font></td>
     <td width="200"></td>
   </tr>
 </table>
 <form action="cubrid.asp" method="post" >
 <table border="1" width="700" cellspacing="0" cellpadding="0" height="45">
```

```
   <tr>
     <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">SERVER IP</font></td>
     <td width="78"  valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">Broker PORT</font></td>
     <td width="148" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB NAME</font></td>
     <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB USER</font></td>
     <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB PASS</font></td>
     <td width="80" height="37" rowspan="4" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED">
       <p><input type="submit" value="Execute" name="B1" tabindex="7"></p></td>
   </tr>
   <tr>
     <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="server_ip" size="20" tabindex="1" maxlength="15"
value="<%=Request("server_ip")%>"></font></td>
     <td width="78"  height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="cas_port" size="15" tabindex="2" maxlength="6"
value="<%=Request("cas_port")%>"></font></td>
     <td width="148" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="db_name" size="20" tabindex="3" maxlength="20"
value="<%=Request("db_name")%>"></font></td>
     <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="db_user" size="15" tabindex="4"
value="<%=Request("db_user")%>"></font></td>
     <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="password" name="db_pass" size="15" tabindex="5"
value="<%=Request("db_pass")%>"></font></td>
   </tr>
   <tr>
     <td width="573" colspan="5" valign="bottom" height="18" bordercolorlight="#FFFFCC"
bgcolor="#DBD7BD"><font size="2">QUERY</font></td>
   </tr>
   <tr>
     <td width="573" colspan="5" height="25" bordercolorlight="#FFFFCC"
bgcolor="#F5F5ED"><textarea rows="3" name="query" cols="92"
tabindex="6"><%=Request("query")%></textarea></td>
   </tr>
  </table>
  </form>
  <hr>
</BODY>
</HTML>
<%
    ' Fetch the DSN and SQL statement.
    strIP = Request( "server_ip" )
    strPort = Request( "cas_port" )
    strUser = Request( "db_user" )
    strPass = Request( "db_pass" )
    strName = Request( "db_name" )
    strQuery = Request( "query" )

if strIP = "" then
   Response.Write "Please enter the SERVER_IP"
        Response.End 'If no IP entered, end the page
    end if
    if strPort = "" then
       Response.Write "Please enter the port number"
        Response.End ' If no port entered, end the page
    end if
    if strUser = "" then
       Response.Write "Please enter the DB_USER"
        Response.End ' If no DB_User entered, end the page
    end if
    if strName = "" then
       Response.Write "Please enter the DB_NAME "
        Response.End ' If no DB_NAME entered, end the page
    end if
    if strQuery = "" then
       Response.Write "Please enter the query you want to check"
        Response.End ' If no Query entered, end the page
```

```
    end if
 ' Create the connection object
  strDsn = "driver={CUBRID Driver};server=" & strIP & ";port=" & strPort & ";uid=" &
strUser & ";pwd=" & strPass & ";db_name=" & strName & ";"
' Connect to DB
Set DBConn = Server.CreateObject("ADODB.Connection")
       DBConn.Open strDsn
    ' Execute SQL
    Set rs = DBConn.Execute( strQuery )
    ' Show message depending on the SQL statement
    if InStr(Ucase(strQuery),"INSERT")>0 then
        Response.Write "The record has been added."
        Response.End
    end if

    if InStr(Ucase(strQuery),"DELETE")>0  then
        Response.Write "The record has been deleted."
        Response.End
    end if

    if InStr(Ucase(strQuery),"UPDATE")>0  then
        Response.Write "The record has been modified."
        Response.End
    end if
%>
<table>
<%
    ' Show the field name
    Response.Write "<tr bgColor=#f3f3f3>"
    For index =0 to ( rs.fields.count-1 )
        Response.Write "<td><b>" & rs.fields(index).name & "</b></td>"
    Next
    Response.Write "</tr>"
    ' Show the field value
    Do While Not rs.EOF
        Response.Write "<tr bgColor=#f3f3f3>"
        For index =0 to ( rs.fields.count-1 )
            Response.Write "<td>" & rs(index) & "</td>"
        Next
        Response.Write "</tr>"

        rs.MoveNext
    Loop
%>
<%
    set  rs = nothing
%>
</table>
```

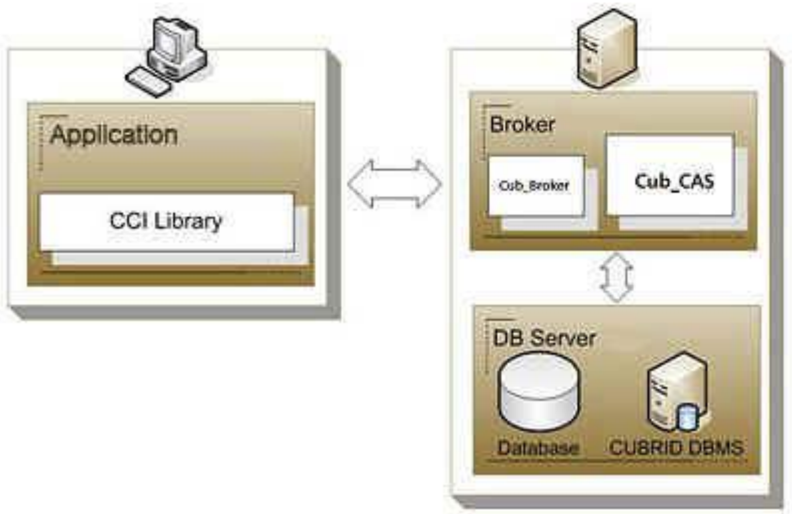
You can check the result of the sample program at http://localhost/aSP/cubrid.asp. When you execute the sample code above, you will get the following output. Enter appropriate values in each field, and then enter the query statement in the Query field. When you click [Run], the query result will be displayed at the lower portion of the page.


CUBRID Query Test Page - Windows Internet Explorer
http://localhost/cubrid.asp
Live Search
CUBRID Query Test Page
Page
Tools
CUBRIDQuery Test
SERVER IP
Broker PORT
DB NAME
DB USER
DB PASS
QUERY
Run
Please enter the SERVER_IP
Local Intranet
100%

Programming with CCI

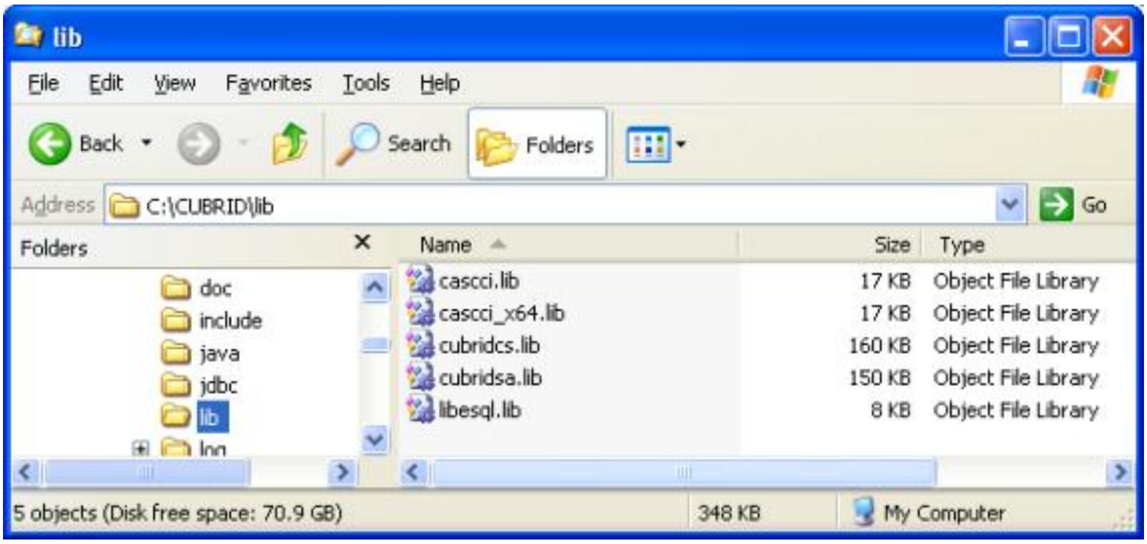
CCI Library

The CCI Library is a C language interface provided by CUBRID. CCI is connected to the application through the Broker, so you can manage it the same way as other interfaces such as JDBC, PHP and ODBC. In fact, CCI provides a foundation to implement PHP, ODBC, Python and, Ruby interfaces.



CCI Installation and Configuration

The CCI library is contained in the CUBRID installation package. The following figure shows where the files are located.



Operating System	Windows	UNIX/Linux
C header file	include/cas_cci.h	include/cas_cci.h
Static library	lib/cascci.lib	lib/libcascci.a
Dynamic library	lib/cascci.lib bin/cascci.dll	lib/libcascci.so

Using CCI

Basic Flow Diagram of the Application Using CCI

To use CUBRID, the following procedures are required for applications using the CCI libraries to execute queries: connection to CAS, query preparation, query execution, response handling, and disconnection. In each process, CCI communicates with the application using connection, query and response handles.

The following flowchart shows the process of the application using CCI and the functions used in each step. See CCI API in the API Reference for more information.

- Opening a database connection handle (related function : cci_connect)
- Getting the request handle for a prepared statement (related function : cci_prepare)
- Binding data to the prepared statement (related function : cci_bind_param)
- Executing the prepared statement (related function : cci_execute)
- Processing the execution result (related function : cci_cursor, cci_fetch, cci_get data, cci_get_result_info)
- Closing the request handle (related function : cci_close_req_handle)
- Closing a database connection handle (related function : cci_disconnect)

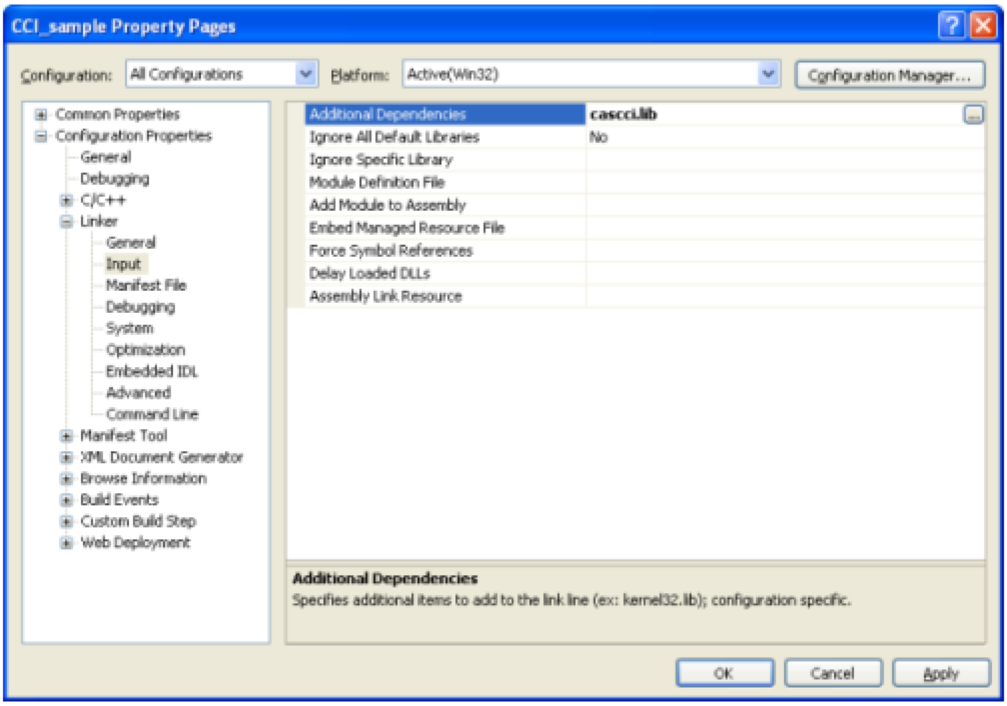
How to use

Once you have created the application using CCI, you should decide, according to its features, whether to execute CCI as a static link or dynamic link before you build it. Determine the library to use by referring to the table in the CCI Installation and Configuration.

The following is an example Makefile to use the dynamic link library on UNIX/Linux:

```
CC=gcc
CFLAGS = -g -Wall -I. -I$CUBRID/include
LDFLAGS = -L$CUBRID/lib -lcascci -lnsl
TEST_OBJS = test.o
EXES = test
all: $(EXES)
test: $(TEST_OBJS)
 $(CC) -o $@ $(TEST_OBJS) $(LDFLAGS)
```

The following is the settings for using the static library on Windows:



CCI Sample

Introduction

The sample program is to create a simple application using CCI through the connection to the **demodb** database deployed by default during the CUBRID installation. Follow the processes of connection to CAS, query preparation, query execution, response handling and disconnection in the sample. The sample is created in a way that uses dynamic links on Linux.

The following is schema information of the **olympic** table in the **demodb** database used in the sample.

```
csql> ;sc olympic
=== <Help: Sechma of a Class> ===
 <Class Name>
    olympic
 <Attributes>
    host_year                 INTEGER NOT NULL
    host_nation               CHARACTER VARYING(40) NOT NULL
    host_city                 CHARACTER VARYING(20) NOT NULL
    opening_date              DATE NOT NULL
    closing_date              DATE NOT NULL
    mascot                    CHARACTER VARYING(20)
    slogan                    CHARACTER VARYING(40)
    introduction              CHARACTER VARYING(1500)

 <Constraints>
    PRIMARY KEY pk_olympic_host_year ON olympic (host_year)
```

Preparation

Make sure that the **demodb** database and the Broker are running before you execute the sample program. You can start the **demodb** database and the Broker by executing the **cubrid** utility.

The following example shows how to run a database server and broker by executing the **cubrid** utility.

```
[tester@testdb ~]$ cubrid server start demodb
@ cubrid master start
++ cubrid master start: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R4.0

++ cubrid server start: success
[tester@testdb ~]$ cubrid broker start
@ cubrid broker start
++ cubrid broker start: success
```

Build

With the program source and the Makefile ready, executing "make" will create an executable file called "test." If you use a static library, there is no need to deploy additional files and the execution will be faster. However, it increases the program size and memory usage. If you use a dynamic library, there will be some performance overhead, but the program size and memory usage can be optimized.

The following is a command line example. It builds the test program using the dynamic library instead of "make" on Linux.

```
cc -o test test.c -I$CUBRID/include -L$CUBRID/lib -lnsl -lcascci
```

Sample Code

```
#include <stdio.h>
#include <cas_cci.h>
char *cci_client_name = "test";
int main (int argc, char *argv[])
{
    int con = 0, req = 0, col_count = 0, res, ind, i;
```

```
    T_CCI_ERROR error;
    T_CCI_COL_INFO *res_col_info;
    T_CCI_SQLX_CMD cmd_type;
    char *buffer, db_ver[16];
    printf("Program started!\n");
    if ((con=cci_connect("localhost", 30000, "demodb", "PUBLIC", ""))<0) {
        printf( "%s(%d): cci_connect fail\n", __FILE__, __LINE__);
        return -1;
    }

    if ((res=cci_get_db_version(con, db_ver, sizeof(db_ver)))<0) {
        printf( "%s(%d): cci_get_db_version fail\n", __FILE__, __LINE__);
        goto handle_error;
    }
    printf("DB Version is %s\n",db_ver);
    if ((req=cci_prepare(con, "select * from event", 0,&error))<0) {
        printf( "%s(%d): cci_prepare fail(%d)\n", __FILE__, __LINE__,error.err_code);
        goto handle_error;
    }
    printf("Prepare ok!(%d)\n",req);
    res_col_info = cci_get_result_info(req, &cmd_type, &col_count);
    if (!res_col_info) {
        printf( "%s(%d): cci_get_result_info fail\n", __FILE__, __LINE__);
        goto handle_error;
    }

    printf("Result column information\n"
           "========================================\n");
    for (i=1; i<=col_count; i++) {
        printf("name:%s  type:%d(precision:%d scale:%d)\n",
            CCI_GET_RESULT_INFO_NAME(res_col_info, i),
            CCI_GET_RESULT_INFO_TYPE(res_col_info, i),
            CCI_GET_RESULT_INFO_PRECISION(res_col_info, i),
            CCI_GET_RESULT_INFO_SCALE(res_col_info, i));
    }
    printf("========================================\n");
    if ((res=cci_execute(req, 0, 0, &error))<0) {
        printf( "%s(%d): cci_execute fail(%d)\n", __FILE__, __LINE__,error.err_code);
        goto handle_error;
    }
    if ((res=cci_fetch_size(req, 100))<0) {
        printf( "%s(%d): cci_fetch_size fail\n", __FILE__, __LINE__);
        goto handle_error;
    }

    while (1) {
        res = cci_cursor(req, 1, CCI_CURSOR_CURRENT, &error);
        if (res == CCI_ER_NO_MORE_DATA) {
            printf("Query END!\n");
            break;
        }
        if (res<0) {
            printf( "%s(%d): cci_cursor fail(%d)\n", __FILE__, __LINE__,error.err_code);
            goto handle_error;
        }

        if ((res=cci_fetch(req, &error))<0) {
            printf( "%s(%d): cci_fetch fail(%d)\n", __FILE__, __LINE__,error.err_code);
            goto handle_error;
        }

        for (i=1; i<=col_count; i++) {
            if ((res=cci_get_data(req, i, CCI_A_TYPE_STR, &buffer, &ind))<0) {
                printf( "%s(%d): cci_get_data fail\n", __FILE__, __LINE__);
                goto handle_error;
            }
            printf("%s \t|", buffer);
        }
        printf("\n");
    }
    if ((res=cci_close_req_handle(req))<0) {
           printf( "%s(%d): cci_close_req_handle fail", __FILE__, __LINE__);
           goto handle_error;
```

```
    }
    if ((res=cci_disconnect(con, &error))<0) {
          printf( "%s(%d): cci_disconnect fail(%d)", __FILE__, __LINE__,error.err_code);
          goto handle_error;
    }
    printf("Program ended!\n");
    return 0;

handle_error:
    if (req > 0)
        cci_close_req_handle(req);
    if (con > 0)
        cci_disconnect(con, &error);
    printf("Program failed!\n");
    return -1;
}
```

CSQL Interpreter

CSQL Interpreter

To execute SQL statements in CUBRID, you need to use either a Graphical User Interface (GUI)-based CUBRID Manager or a console-based CSQL Interpreter.

CSQL is an application that allows users to use SQL statements through a command-driven interface. This section briefly explains how to use the CSQL Interpreter and associated commands.

- Introduction to the CSQL Interpreter
- Running CSQL
- Session Commands

Introduction to the CSQL Interpreter

A Tool for SQL

The CSQL Interpreter is an application installed with CUBRID that allows you to execute in an interactive or batch mode and viewing query results. The CSQL Interpreter has a command-line interface. With this, you can save SQL statements together with their results to a file for a later use.

The CSQL Interpreter provides the best and easiest way to use CUBRID. You can develop database applications with various APIs (e.g. JDBC, ODBC, PHP, CCI, etc.; you can use the CUBRID Manager, which is a management and query tool provided by CUBRID. With the CSQL Interpreter, users can create and retrieve data in a terminal-based environment.

The CSQL Interpreter directly connects to a CUBRID database and executes various tasks using SQL statements. Using the CSQL Interpreter, you can:

- Retrieve, update and delete data in a database by using SQL statements
- Execute external shell commands
- Save or print query results
- Create and execute SQL script files
- Select table schema
- Retrieve or modify parameters of the database server system
- Retrieve database information (e.g. schema, triggers, queued triggers, workspaces, locks, and statistics)

A Tool for DBA

A database administrator (**DBA**) performs administrative tasks by using various administrative utilities provided by CUBRID; a terminal-based interface of CSQL Interpreter is an environment where **DBA** executes administrative tasks.

It is also possible to run the CSQL Interpreter in a standalone mode. In this mode, the CSQL Interpreter directly accesses database files and executes commands including server process properties. That is, SQL statements can be executed to a database without running a separate database server process. The CSQL Interpreter is a powerful tool that allows you to use the database only with a **csql** utility, without any other applications such as the Database Server or the Brokers.

Executing CSQL

CSQL Execution Mode

Interactive Mode

With CSQL Interpreter, you can enter and execute SQL statements to handle schema and data in the database. Enter statements in a prompt that appears when running the **csql** utility. After executing the statements, the results are listed in the next line. This is called the interactive mode.

Batch Mode

You can save SQL statements in a file and execute them later to have the **csql** utility read the file. This is called the batch mode. For more information on the batch mode, see CSQL Startup Options.

Standalone Mode

In the standalone mode, CSQL Interpreter directly accesses database files and executes commands including server process functions. That is, SQL statements can be sent and executed to a database without a separate database server process running for the task. Since the standalone mode allows only one user access at a given time, it is suitable for management tasks by Database Administrators (**DBAs**).

Client/Server Mode

CSQL Interpreter usually operates as a client process and accesses the server process.

Using CSQL (Syntax)

Connecting to Local Host

Description

Execute the CSQL Interpreter using the **csql** utility. You can set options as needed. To set the options, specify the name of the database to connect to as a parameter. The following is a **csql** utility statement to access the database on a local server:

Syntax

```
csql [ options ] database_name
```

Connecting to Remote Host

Descripton

The following is a **csql** utility statement to access the database on a remote host:

Syntax

```
csql [ options ] database_name@remote_host_name
```

Make sure that the following conditions are met before you run the CSQL Interpreter on a remote host.

- The CUBRID installed on the remote host must be the same version as the one on the local host.
- The port number used by the master process on the remote host must be identical to the one on the local host.
- You must access the remote host in a client/server mode using the **-C** option.

Example

The following is an example statement that accesses the **demodb** database on the remote host with the IP address 192.168.1.3 and calls the **csql** utility.

```
csql -C demodb@192.168.1.3
```

CSQL Startup Options

To display the option list in the prompt, execute the **csql** utility without specifying the database name as follows:

```
 % csql
interactive SQL utility, version R4.0
usage: csql [OPTION] database-name valid options:
  -S, --SA-mode                standalone mode execution
  -C, --CS-mode                client-server mode execution
  -u, --user=ARG               alternate user name
  -p, --pasword=ARG            password string, give "" for none
  -e, --error-continue         don't exit on statement error
  -i, --input-file=ARG         input-file-name
  -o, --output-file=ARG        output-file-name
  -s  --single-line            single line oriented execution
  -c, --command=ARG            CSQL-commands
  -l, --line-output            display each value in a line
  -r, --read-only              read-only mode
      --no-auto-commit         disable auto commit mode execution
      --no-pager               do not use pager
      --no-single-line         turn off single line oriented execution
```

Options

The following table lists the options that can be issued with the **csql** utility.

Options	Description
-S	Executes the csql utility in a standalone mode.
-C	Executes the csql utility in a client/server mode.
-u *user_name*	Specifies the user that tries to access the database. The default value is **PUBLIC**.
-p *password*	Specifies the password of the user that tries to access the database (if any).
-e	Continues the session even when an error occurs.
-i *input_file*	Executes the csql utility in a batch mode. The *input_file* parameter is the file name where SQL statements are saved.
-o *output_file*	Saves a result of the statement execution in the specified *output_file* without displaying it on the screen.
-s	Executes multiple SQL statements one by one in the file where they are saved consecutively. Multiple SQL statements are separated by semicolons (;). If this option is not specified, the default operation is performed.
-c "CSQL commands"	Executes SQL statements directly from the prompt. To use this option, enclose the SQL statement to execute in double quotes.
-l	Displays the query results in a line format instead of a column. By default, the results will be displayed in a column format.
-r	Connects to a database in read-only mode.
--no-auto-commit	Configures the auto-commit mode of the CSQL Interpreter to OFF.
--no-pager	Displays the results of the query performed by the CSQL Interpreter at once instead of page-by-page.

--no-single-line	Executes multiple SQL statements at once by using **;xr** or **;r** session command.

Executing in a standalone mode (-S)

The following is an example to connect to the **demodb** database in a standalone mode and execute the **csql** utility with the **-S** option. When you want to use the **demodb** database exclusively, use the **-S** option.

```
csql -S demodb
```

Executing in a client/server mode (-C)

The following is an example to connect to the **demodb** database in a client/server mode and execute the **csql** utility with the **-C** option. In an environment where multiple clients connect to the **demodb** database, use the **-C** option. Even when you connect to a database on a remote host in a client/server mode, the error log created during the **csql** utility execution will be saved in the **cub_client.err** file on the local host.

```
csql -C demodb
```

Specifying the name of the input file to use in a batch mode (-i)

The following is an example to specify the name of the input file that will be used in a batch mode with the **-i** option. In the **infile** file, more than one SQL statement are saved. Without the **-i** option specified, the CSQL Interpreter will run in an interactive mode.

```
csql -i infile demodb
```

Specifying the output file to save the execution results (-o)

The following is an example to save the execution results to the specified file instead of displaying on the screen with the **-o** option. This option is useful when you want to retrieve the results of the query performed by the CSQL Interpreter at a later time.

```
csql -o outfile demodb
```

Specifying the user name (-u)

The following is an example to specify the name of the user that will connect to the specified database with the **-u** option. If the **-u** option is not specified, **PUBLIC** that has the lowest level of authorization will be specified as a user. If the user name is not valid, an error message is displayed and the **csql** utility is terminated. If there is a password for the user name you specify, you will be prompted to enter the password.

```
csql -u DBA demodb
```

Specifying the user password (-p)

The following is an example to enter the password of the user specified with the **-p** option. Especially since there is no prompt to enter a password for the user you specify in a batch mode, you must enter the password using the **-p** option. When you enter an incorrect password, an error message is displayed and the **csql** utility is terminated.

```
csql -u DBA -p *** demodb
```

Executing SQL statements one by one (-s)

The following is an example to execute SQL statements one by one with the **-s** option. Use this option when you want to allocate less memory for the query execution. Multiple SQL statements are separated by semicolons (;). If this option is not specified, the default operation is performed.

```
csql -s -i infile demodb
```

Executing SQL statements directly from the shell (-c)

The following is an example to execute more than one SQL statement from the shell with the **-c** option. Multiple statements are separated by semicolons (;).

```
csql -c "select * from olympic;select * from stadium" demodb
```

Displaying the results in a line format (-l)

The following is an example to display the execution results of the SQL statement in a line format with the **-l** option. The execution results will be output in a column format if the **-l** option is not specified.

```
csql -l demodb
```

Continuing the execution even with an error (-e)

The following is an example to continue to execute subsequent SQL statements even when a syntax error or a runtime error occurs in a previous SQL statement by using the **-e** option. When there is an error in the SQL statement, the database will be terminated even though the **-e** option is specified.

```
csql -e demodb
```

Connecting to a database in read-only mode (-r)

The following is an example to connect to a database in read-only mode by using the **-r** option. Creating a table or manipulating data is not allowed; only retrieving data is allowed.

```
csql -r demodb
```

No auto-commit mode (--no-auto-commit)

The following is an example to stop the auto-commit mode with the **--no-auto-commit** option. If you don't configure **--no-auto-commit** option, the CSQL Interpreter runs in an auto-commit mode by default, and the SQL statement is committed automatically at every execution. Executing the **;AUtocommit** session command after starting the CSQL Interpreter will also have the same result.

```
csql --no-auto-commit demodb
```

Displaying all the execution results at once (--no-pager)

The following is an example to display the execution results by the CSQL Interpreter at once instead of page-by-page with the **--no-pager** option. The results will be output page-by-page if **--no-pager** option is not specified.

```
csql --no-pager demodb
```

Executing all SQL statements at once (--no single-line)

The following is an example to execute all SQL statements at once by using **;xr** or **;r** session command. If you do not specify this option, SQL statements are executed without **;xr** or **;r** session command.

```
csql --no-pager demodb
```

Session Commands

In addition to SQL statements, CSQL Interpreter provides special commands allowing you to control the Interpreter. These commands are called session commands. All the session commands must start with a semicolon (;).

Session Commands

Enter the **;help** command to display a list of the session commands available in the CSQL Interpreter. Note that only the uppercase letters of each session command are required to make the CSQL Interpreter to recognize it. Session commands are not case-sensitive.

```
CUBRID SQL Interpreter
Type `;help' for help messages.
csql> ;help
=== <Help: Session Command Summary> ===
   All session commands should be prefixed by `;' and only blanks/tabs
   can precede the prefix. Capitalized characters represent the minimum
   abbreviation that you need to enter to execute the specified command.
   ;REAd   [<file-name>]       - read a file into command buffer.
   ;Write  [<file-name>]       - (over)write command buffer into a file.
   ;APpend [<file-name>]       - append command buffer into a file.
   ;PRINT                      - print command buffer.
   ;SHELL                      - invoke shell.
   ;CD                         - change current working directory.
   ;EXit                       - exit program.
   ;CLear                      - clear command buffer.
   ;EDIT                       - invoke system editor with command buffer.
   ;List                       - display the content of command buffer.
   ;RUn                        - execute sql in command buffer.
   ;Xrun                       - execute sql in command buffer, and clear the command
buffer.
   ;COmmit                     - commit the current transaction.
   ;ROllback                   - roll back the current transaction.
   ;AUtocommit [ON|OFF]        - enable/disable auto commit mode.
   ;CHeckpoint                 - issue checkpoint.
   ;Killtran                   - kill transaction.
   ;REStart                    - restart database.
   ;SHELL_Cmd  [shell-cmd]     - set default shell, editor, print and pager
   ;EDITOR_Cmd [editor-cmd]       command to new one, or display the current
   ;PRINT_Cmd  [print-cmd]        one, respectively.
   ;PAger_cmd  [pager-cmd]
   ;DATE                       - display the local time, date.
   ;DATAbase                   - display the name of database being accessed.
   ;SChema class-name          - display schema information of a class.
   ;SYntax [sql-cmd-name]      - display syntax of a command.
   ;TRigger [`*'|trigger-name] - display trigger definition.
   ;Get system_parameter       - get the value of a system parameter.
   ;SEt system_parameter=value - set the value of a system parameter.
   ;PLan [simple/detail/off]   - show query execution plan.
   ;Info <command>             - display internal information.
   ;TIme [ON/OFF]              - enable/disable to display the query execution time.
   ;HISTORYList                - display list of the executed queries.
   ;HISTORYRead <history_num>  - read entry on the history number into command buffer.
   ;HElp                       - display this help message.
csql>
```

Options

Reading SQL statements from a file (;REAd)

The **;REAd** command reads the contents of a file into the command buffer. This command is used to execute SQL commands saved in the specified file. To view the contents of the file loaded into the buffer, use the **;List** command.

```
csql> ;rea nation.sql
The file has been read into the command buffer.
csql> ;list
insert into "sport_event" ("event_code", "event_name", "gender_type", "num_player") values
```

```
(20001, 'Archery Individual', 'M', 1);
insert into "sport_event" ("event_code", "event_name", "gender_type", "num_player") values
20002, 'Archery Individual', 'W', 1);
....
```

Saving SQL statements into a file (;Write)

The **;Write** command saves the contents of the command buffer into a file. This command is used to save SQL commands that you entered or modified in the CSQL Interpreter.

```
csql> ;w outfile
Command buffer has been saved.
```

Appending to a file (;APpend)

This command appends the contents of the current command buffer to an **outfile** file.

```
csql> ;ap outfile
Command buffer has been saved.
```

Executing a shell command (;SHELL)

The **;SHELL** session command calls an external shell. Starts a new shell in the environment where the CSQL Interpreter is running. It returns to the CSQL Interpreter when the shell terminates. If the shell command to execute with the **;SHELL_Cmd** command has been specified, it starts the shell, executes the specified command, and returns to the CSQL Interpreter.

```
csql> ;shell
% Is -al
total 2088
drwxr-xr-x 16 DBA cubrid   4096 Jul 29 16:51 .
drwxr-xr-x  6 DBA cubrid   4096 Jul 29 16:17 ..
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 02:49 audit
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 16:17 bin
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 16:17 conf
drwxr-xr-x  4 DBA cubrid   4096 Jul 29 16:14 cubridmanager
% exit
csql>
```

Registering a shell command (;SHELL_Cmd)

The **;SHELL_Cmd** command registers a shell command to execute with the **SHELL** session command. As shown in the example below, enter the **;shell** command to execute the registered command.

```
csql> ;shell_c ls -la
csql> ;shell
total 2088
drwxr-xr-x 16 DBA cubrid   4096 Jul 29 16:51 .
drwxr-xr-x  6 DBA cubrid   4096 Jul 29 16:17 ..
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 02:49 audit
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 16:17 bin
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 16:17 conf
drwxr-xr-x  4 DBA cubrid   4096 Jul 29 16:14 cubridmanager
csql>
```

Changing the current working directory (;CD)

This command changes the current working directory where the CSQL Interpreter is running to the specified directory. If you don't specify the path, the directory will be changed to the home directory.

```
csql> ;cd /home1/DBA/CUBRID
Current directory changed to  /home1/DBA/CUBRID.
```

Exiting the CSQL Interpreter (;EXit)

This command exits the CSQL Interpreter.

```
csql> ;ex
```

Clearing the command buffer (;CLear)

The **;CLear** session command clears the contents of the command buffer.

```
csql> ;cl
csql> ;list
```

Displaying the contents of the command buffer (;List)

The **;List** session command lists the contents of the command buffer that have been entered or modified. The command buffer can be modified by **;READ** or **;Edit** command.

```
csql> ;l
```

Executing SQL statements (;RUn)

This command executes SQL statements in the command buffer. Unlike the **;Xrun** session command described below, the buffer will not be cleared even after the query execution.

```
csql> ;ru
```

Clearing the command buffer after executing the SQL statement (;Xrun)

This command executes SQL statements in the command buffer. The buffer will be cleared after the query execution.

```
csql> ;x
```

Committing transaction (;COmmit)

This command commits the current transaction. You must enter a commit command explicitly if it is not in auto-commit mode. In auto-commit mode, transactions are automatically committed whenever SQL is executed.

```
csql> ;co
Current transaction has been committed.
```

Rolling back transaction (;ROllback)

This command rolls back the current transaction. Like a commit command (**;COmmit**), it must enter a rollback command explicitly if it is not in auto-commit mode (**OFF**).

```
csql> ;ro
Current transaction has been rolled back.
```

Setting the auto-commit mode (;AUtocommit)

This command sets auto-commit mode to **ON** or **OFF**. If any value is not specified, current configured value is applied by default. The default value is **ON**.

```
csql> ;au off
AUTOCOMMIT IS OFF
```

CHeckpoint Execution (;CHeckpoint)

This command executes the checkpoint within the CSQL session. This command can only be executed when a DBA group member, who is specified for the custom option (**-u** *user_name*), connects to the CSQL interpreter in system administrator mode (**--sysadm**).

Checkpoint is an operation of flushing log files (dirty pages) from the current data buffer to disks. You can also change the checkpoint interval using a command (**;set** *parameter_name* value) to set the parameter values in the CSQL session. You can see the examples of the parameter related to the checkpoint execution interval (**checkpoint_interval_in_mins** and **checkpoint_every_npages**). For more information, see Logging-related Parameters.

```
csql> ;ch
Checkpoint has been issued.
```

Transaction Monitoring Or Termination (;Killtran)

This command checks the transaction status information or terminates a specific transaction in the CSQL session. This command prints out the status information of all transactions on the screen if a parameter is omitted it terminates the transaction if a specific transaction ID is specified for the parameter. It can only be executed when a DBA group member, who is specified for the custom option (**-u** *user_name*), connects to the CSQL interpreter in system administrator mode (**--sysadm**).

```
csql> ;k
Tran index      User name      Host name      Process id      Program name
-------------------------------------------------------------------------------
      1(+)            dba      myhost              664           cub_cas
      2(+)            dba      myhost             6700              csql
      3(+)            dba      myhost             2188           cub_cas
      4(+)            dba      myhost              696              csql
      5(+)         public      myhost             6944              csql

csql> ;k 3
The specified transaction has been killed.
```

Restarting database (;REStart)

A command that tries to reconnect to the target database in a CSQL session. Note that when you execute the CSQL interpreter in CS (client/server) mode, it will be disconnected from the server. When the connection to the server is lost due to a HA failure and failover to another server occurs, this command is particularly useful in connecting to the switched server while maintaining the current session.

```
csql> ;res
The database has been restarted.
```

Displaying the current date (;DATE)

The **;DATE** command displays the current date and time in the CSQL Interpreter.

```
csql> ;date
     Tue July 29 18:58:12 KST 2008
```

Displaying the database informatio (;DATAbase)

This command displays the database name and host name where the CSQL Interpreter is working. If the database is running, the HA mode (one of those followings: active, standby, or maintenance) will be displayed as well.

```
csql> ;data
     demodb@localhost (active)
```

Displaying schema information of a class (;SChema)

The **;SChema** session command displays schema information of the specified table. The information includes the table name, its column name and constraints.

```
csql> ;sc event
=== <Help: Schema of a Class> ===
 <Class Name>
     event
 <Attributes>
     code           INTEGER NOT NULL
     sports         CHARACTER VARYING(50)
     name           CHARACTER VARYING(50)
     gender         CHARACTER(1)
     players        INTEGER
 <Constraints>
     PRIMARY KEY pk_event_event_code ON event (code)
```

Displaying syntax (;SYntax)

This command displays the syntax of the SQL statement specified. If there is no specific syntax specified, all the syntaxes defined and their rules will be displayed.

```
csql> ;sy alter
=== <Help: Command Syntax> ===
 <Name>
   ALTER
 <Description>
Change the definition of a class or virtual class.
 <Syntax>
<alter> ::= ALTER [ <class_type> ] <class_name> <alter_clause> ;
```

```
<class_type> ::= CLASS | TABLE | VCLASS | VIEW
<alter_clause> ::= ADD <alter_add> [ INHERIT <resolution_comma_list> ] |
                   DROP <alter_drop> [ INHERIT <resolution_comma_list> ] |
                   RENAME <alter_rename> [ INHERIT <resolution_comma_list> ] |
>                   CHANGE <alter_change> |
                   INHERIT <resolution_comma_list>
<alter_add> ::= [ ATTRIBUTE | COLUMN ] <class_element_comma_list> |
                CLASS ATTRIBUTE <attribute_definition_comma_list> |
                FILE <file_name_comma_list> |
                METHOD <method_definition_comma_list> |
                QUERY <select_statement> |
                SUPERCLASS <class_name_comma_list>
......
```

Displaying the trigger (;TRriger)

This command searches and displays the trigger specified. If there is no trigger name specified, all the triggers defined will be displayed.

```
csql> ;tr
=== <Help: All Triggers> ===
     trig_delete_contents
```

Checking the parameter value(;Get)

You can check the parameter value currently set in the CSQL Interpreter using the **;Get** session command. An error occurs if the parameter name specified is incorrect.

```
csql> ;g isolation_level
=== Get Param Input ===
isolation_level=4
```

Setting the parameter value (;SEt)

You can use the **;Set** session command to set a specific parameter value. Note that changeable parameter values are only can be changed. To change the server parameter values, you must have DBA authorization. For information on list of changeable parameters, see cubrid_broker.conf Configuration File and Default Parameters.

```
csql> ;se block_ddl_statement=1
=== Set Param Input ===
block_ddl_statement=1
-- Dynamically change the log_max_archives value in the csql accessed by dba account
csql>;se log_max_archives=5
```

Setting the view level of executing query plan (;PLan)

You can use the **;PLan** session command to set the view level of executing query plan the level is composed of **simple**, **detail**, and **off**. Each command refers to the following:

- **off** : Not displaying the query execution plan
- **simple** : Displaying the query execution plan in simple version (OPT LEVEL=257)
- **detail** : Displaying the query execution plan in detailed version (OPT LEVEL=513)

Displaying information (;Info)

The **;Info** session command allows you to check information such as schema, triggers, the working environment, locks and statistics.

```
csql> ;i lock
*** Lock Table Dump ***
 Lock Escalation at = 100000, Run Deadlock interval = 1
Transaction (index  0, unknown, unknown@unknown|-1)
Isolation REPEATABLE CLASSES AND READ UNCOMMITTED INSTANCES
State TRAN_ACTIVE
Timeout_period -1
......
```

Outputting statistics information of server processing (;.Hist)

This command shows the statistics information of server processing. The information is collected after this command is entered. Therefore, the execution commands such as **;.dump_hist** or **;.x** must be entered to output the statistics information

This command is executable while the **communication_histogram** parameter in the **cubrid.conf** file is set to **yes**. You can also view this information by using the **cubrid statdump** utility. Following options are provided for this session command.

- **on** : Starts collecting statistics information for the current connection.
- **off** : Stops collecting statistics information of server.

This example shows the server statistics information for current connection. For information on specific items, see Outputting Statistics Information of Server.

```
csql> ;.hist on
csql> ;.x
Histogram of client requests:
Name                     Rcount         Sent size         Recv size      Server time
 No server requests made
 *** CLIENT EXECUTION STATISTICS ***
System CPU (sec)              =      0
User CPU (sec)                =      0
Elapsed (sec)                 =      20
 *** SERVER EXECUTION STATISTICS ***
Num_file_creates              =      0
Num_file_removes              =      0
Num_file_ioreads              =      0
Num_file_iowrites             =      0
Num_file_iosynches            =      0
Num_data_page_fetches         =      56
Num_data_page_dirties         =      14
Num_data_page_ioreads         =      0
Num_data_page_iowrites        =      0
Num_data_page_victims         =      0
Num_data_page_iowrites_for_replacement =      0
Num_log_page_ioreads          =      0
Num_log_page_iowrites         =      0
Num_log_append_records        =      0
Num_log_archives              =      0
Num_log_checkpoints           =      0
Num_log_wals                  =      0
Num_page_locks_acquired       =      2
Num_object_locks_acquired     =      2
Num_page_locks_converted      =      0
Num_object_locks_converted    =      0
Num_page_locks_re-requested   =      0
Num_object_locks_re-requested =      1
Num_page_locks_waits          =      0
Num_object_locks_waits        =      0
Num_tran_commits              =      1
Num_tran_rollbacks            =      0
Num_tran_savepoints           =      0
Num_tran_start_topops         =      3
Num_tran_end_topops           =      3
Num_tran_interrupts           =      0
Num_btree_inserts             =      0
Num_btree_deletes             =      0
Num_btree_updates             =      0
Num_btree_covered             =      0
Num_btree_noncovered          =      0
Num_btree_resumes             =      0
Num_query_selects             =      1
Num_query_inserts             =      0
Num_query_deletes             =      0
Num_query_updates             =      0
Num_query_sscans              =      1
Num_query_iscans              =      0
Num_query_lscans              =      0
Num_query_setscans            =      0
Num_query_methscans           =      0
Num_query_nljoins             =      0
```

```
Num_query_mjoins              =      0
Num_query_objfetches          =      0
Num_network_requests          =      8
Num_adaptive_flush_pages      =      0
Num_adaptive_flush_log_pages  =      0
Num_adaptive_flush_max_pages  =      0

 *** OTHER STATISTICS ***
Data_page_buffer_hit_ratio    =   100.00
csql> ;.h off
```

Displaying query execution time (;TIme)

The **;TIme** session command can be set to display the elapsed time to execute the query. It can be set to **ON** or **OFF**. The current setting is displayed if there is no value specified. csql> ;ti ON

```
csql> ;ti
TIME IS ON
```

Displaying query history (;HISTORYList)

This command displays the list that contains previously executed commands (input) and their history numbers.

```
csql> ;historyl
----< 1 >----
select * from nation;
----< 2 >----
select * from athlete;
```

Reading input with the specified history number into the buffer (;HISTORYRead)

You can use **;HISTORYRead** session command to read input with history number in the **;HISTORYList** list into the command buffer. You can enter **;ru** or **;x** directly because it has the same effect as when you enter SQL statements directly.

```
csql> ;historyr 1
```

Calling the default editor (;EDIT)

This command calls the specified editor. The default editor is **vi** in Linux environment **Notepad** in Windows environment. Use **;EDITOR_Cmd** command to specify a different editor.

```
csql> ;edit
```

Specifying the editor (;EDITOR_Cmd)

This command specifies the editor to be used with **;EDIT** session command. As shown in the example below, you can specify other editor (ex: **emacs**) which is installed in the system.

```
csql> ;editor_c emacs
csql> ;edit
```

CUBRID SQL Guide

CUBRID SQL Guide

This chapter describes SQL syntax such as data types, functions and operators, data retrieval or table manipulation. You can also find SQL statements used for index, trigger, partition, serial and changing user information.

The main topics covered in this chapter are as follows:

- Glossary
- Comment
- Identifier
- Reserved words
- Data types
- Table definition
- Index definition
- VIEW
- SERIAL
- Operators and functions
- Data retrieval and manipluation
- Query optimization
- Triggers (TRIGGER)
- Java stored functions/procedures
- Methods
- Partitions
- Class Inheritance
- Class Conflict Resolution
- CUBRID System Catalog

Glossary

CUBRID is an object-relational database management system (ORDBMS), which supports object-oriented concepts such as inheritance. In this manual, relational database terminology is also used along with object-oriented terminology for better understanding. Object-oriented terminology such as class, instance and attribute is used to describe concepts including inheritance, and relational database terminology is mainly used to describe common SQL syntax.

The following table provides the summary:

Relational Database	CUBRID
table	class, table
column	attribute, column
record	instance, record
data type	domain, data type

Comment

The CSQL Interpreter is a SQL-style method; the SQL-style comment starts with the double dashes (--) and the comment line after the double dashes is regarded as comment. Additionally, it supports C++ style, which start with double slashes (//), and C-style, which starts and ends with '/*' and '*/' respectively.

The following are examples of comments supported in the CSQL Interpreter.

Example

- --

```
-- This is a SQL-style comment.
```

- //

```
This is a C++ style comment.
```

- /* */

```
/* This is a C-style comment.*/
/* This is an example to use two lines
as comment by using the C-style. */
```

Identifier

Guidelines for Creating Identifiers

The guidelines for creating identifiers in the CSQL Interpreter are as follows:

- An identifier must begin with a letter it must not begin with a number or a symbol.
- It is not case-sensitive.
- CUBRID keywords are not allowed.

< *identifier*>

```
:: = < identifier_letter> [ { < other_identifier>   } ]
```

< *identifier_letter*>

```
:: = < upper_case_letter>
  | < lower_case_letter>
```

< *other_identifier*>

```
:: = < identifier_letter>
  | < digit>
  | _
  | #
```

< *digit*>

```
:: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

< *upper_case_letter*>

```
:: = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P| Q | R | S | T | U | V
| W | X | Y | Z
```

< *lower_case_letter*>

```
:: = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p| q | r | s | t | u | v
| w | x | y | z
```

Legal Identifiers

Beginning with a Letter

An identifier must begin with a letter. All other special characters except operator characters are allowed. The following are examples of legal identifiers.

```
a
a_b
ssn#
fg%
this_is_an_example_#%%#
```

Enclosing in Double Quotes, Square Brackets, or Backtick Symbol

Identifiers or a reserved keywords shown as below are not allowed' however, if they are enclosed in in double quotes, square brackets, or backtick symbol, they are allowed as an exception. Especially, the double quotations can be used as a symbol enclosing identifiers when the **ansi_quote** parameter is set to **yes**. If this value is set to **no**, double quotations are used as a symbol enclosing character strings. The followings are examples of legal identifiers.

```
" select"
" @lowcost"
" low cost"
" abc" " def"
[position]
```

Illegal Identifiers

Beginning with special characters or numbers

An identifier starting with a special character or a number is not allowed. As an exception, a underline (_) and a sharp symbol (#) are allowed for the first character.

```
_a
#ack
%nums
2fer
88abs
```

An identifier containing a space

An identifier that a space within characters is not allowed.

```
coll tl
```

An identifier containing operator special characters

An identifier which contains operator special characters (+, -, *, /, %, ||, !, < , > , =, |, ^, & , ~) is not allowed.

```
col+
col~
col& &
```

Reserved Words

The following keywords are previously reserved as a command, a function name or a type name in CUBRID. You are restricted to use these words for a class name, an attribute name, a variable name. Note than these reserved keywords can be used an identifier when they are enclosed in double quotes, square brackets, or backtick symbol (`).

ABSOLUTE	ACTION	ADD
ADD_MONTHS	AFTER	ALIAS
ALL	ALLOCATE	ALTER
AND	ANY	ARE
AS	ASC	ASSERTION
ASYNC	AT	ATTACH
ATTRIBUTE	AVG	
BEFORE	BETWEEN	BIGINT
BIT	BIT_LENGTH	BLOB
BOOLEAN	BOTH	BREADTH
BY		
CALL	CASCADE	CASCADED
CASE	CAST	CATALOG
CHANGE	CHAR	CHARACTER
CHECK	CLASS	CLASSES
CLOB	CLOSE	CLUSTER
COALESCE	COLLATE	COLLATION
COLUMN	COMMIT	COMPLETION
CONNECT	CONNECT_BY_ISCYCLE	CONNECT_BY_ISLEAF
CONNECTION_BY_ROOT	CONNECTION	CONSTRAINT
CONSTRAINTS	CONTINUE	CONVERT
CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE
CURRENT_DATETIME	CURRENT_TIME	CURRENT_TIMESTAMP
CURRENT_USER	CURSOR	CYCLE
DATA	DATA_TYPE	DATABASE
DATE	DATETIME	DAY
DAY_HOUR	DAY_MILLISECOND	DAY_MINUTE
DAY_SECOND	DEALLOCATE	DEC
DECIMAL	DECLARE	DEFAULT
DEFERRABLE	DEFERRED	DELETE
DEPTH	DESC	DESCRIBE
DESCRIPTOR	DIAGNOSTICS	DICTIONARY
DIFFERENCE	DISCONNECT	DISTINCT

DISTINCTROW	DIV	DO
DOMAIN	DOUBLE	DUPLICATE
DROP		
EACH	ELSE	ELSEIF
END	EQUALS	ESCAPE
EVALUATE	EXCEPT	EXCEPTION
EXCLUDE	EXEC	EXECUTE
EXISTS	EXTERNAL	EXTRACT
FALSE	FETCH	FILE
FIRST	FLOAT	FOR
FOREIGN	FOUND	FROM
FULL	FUNCTION	
GENERAL	GET	GLOBAL
GO	GOTO	GRANT
GROUP	HAVING	HOUR
HOUR_MILLISECOND	HOUR_MINUTE	HOUR_SECOND
IDENTITY	IF	IGNORE
IMMEDIATE	IN	INDEX
INDICATOR	INHERIT	INITIALLY
INNER	INOUT	INPUT
INSERT	INT	INTEGER
INTERSECT	INTERSECTION	INTERVAL
INTO	IS	ISOLATION
JOIN		
KEY		
LANGUAGE	LAST	LDB
LEADING	LEAVE	LEFT
LESS	LEVEL	LIKE
LIMIT	LIST	LOCAL
LOCAL_TRANSACTION_ID	LOCALTIME	LOCALTIMESTAMP
LOOP	LOWER	
MATCH	MAX	METHOD
MILLISECOND	MIN	MINUTE
MINUTE_MILLISECOND	MINUTE_SECOND	MOD
MODIFY	MODULE	MONETARY
MONTH	MULTISET	MULTISET_OF
NA	NAMES	NATIONAL
NATURAL	NCHAR	NEXT
NO	NONE	NOT
NULL	NULLIF	NUMERIC

OBJECT	OCTET_LENGTH	OF
OFF	OID	ON
ONLY	OPEN	OPERATION
OPERATORS	OPTIMIZATION	OPTION
OR	ORDER	OTHERS
OUT	OUTER	OUTPUT
OVERLAPS		
PARAMETERS	PARTIAL	PENDANT
POSITION	PRECISION	PREORDER
PREPARE	PRESERVE	PRIMARY
PRIOR	PRIVATE	PRIVILEGES
PROCEDURE	PROTECTED	PROXY
QUERY		
READ	REAL	RECURSIVE
REF	REFERENCES	REFERENCING
REGISTER	RELATIVE	RENAME
REPLACE	RESIGNAL	RESTRICT
RETURN	RETURNS	REVOKE
RIGHT	ROLE	ROLLBACK
ROLLUP	ROUTINE	ROW
ROWNUM	ROWS	
SAVEPOINT	SCHEMA	SCOPE
SCROLL	SEARCH	SECOND
SECOND_MILLISECOND	SECTION	SELECT
SENSITIVE	SEQUENCE	SEQUENCE_OF
SERIALIZABLE	SESSION	SESSION_USER
SET	SET_OF	SETEQ
SHARED	SIBLINGS	SIGNAL
SIMILAR	SIZE	SMALLINT
SOME	SQL	SQLCODE
SQLERROR	SQLEXCEPTION	SQLSTATE
SQLWARNING	STATISTICS	STRING
STRUCTURE	SUBCLASS	SUBSET
SUBSETEQ	SUBSTRING	SUM
SUPERCLASS	SUPERSET	SUPERSETEQ
SYS_CONNECT_BY_PATH	SYS_DATE	SYS_DATETIME
SYS_TIME	SYS_TIMESTAMP	SYS_USER
SYSDATE	SYSDATETIME	SYSTEM_USER
SYSTIME		
TABLE	TEMPORARY	TEST

THEN	THERE	TIME
TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRANSACTION
TRANSLATE	TRANSLATION	TRIGGER
TRIM	TRUE	TRUNCATE
TYPE		
UNDER	UNION	UNIQUE
UNKNOWN	UPDATE	UPPER
USAGE	USE	USER
USING	UTIME	
VALUE	VALUES	VARCHAR
VARIABLE	VARYING	VCLASS
VIEW	VIRTUAL	VISIBLE
WAIT	WHEN	WHENEVER
WHERE	WHILE	WITH
WITHOUT	WORK	WRITE
XOR		
YEAR	YEAR_MONTH	
ZONE		

Data Types

Numeric Types

Definition and Characteristics

Definition

CUBRID supports the following numeric data types to store integers or real numbers.

Numeric Types Supported by CUBRID

Type	Bytes	Mix	Max	Exact/approx.
SHORT SMALLINT	2	-32,768	+32,767	exact numeric
INT INTEGER	4	-2,147,483,648	+2,147,483,647	exact numeric
BIGINT	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807	exact numeric
NUMERIC DECIMAL	16	precision p : 1 scale s : 0	precision p : 38 scale s : 38	exact numeric
FLOAT REAL	4	-3.402823466E+38 (ANSI/IEEE 754-1985 standard)	+3.402823466E+38 (ANSI/IEEE 754-1985 standard)	approximate numeric floating point : 7
DOUBLE DOUBLE PRECISION	8	-1.7976931348623157E+308 ANSI/IEEE 754-1985 standard)	+1.7976931348623157E+308(ANSI/IEEE 754-1985 standard)	approximate numeric floating point : 15
MONETARY	12	-3.402823466E+38	+3.402823466E+38	approximate numeric

Numeric data types are divided into exact and approximate types. Exact numeric data types (**SMALLINT**, **INT**, **BIGINT**, **NUMERIC**) are used for numbers whose values must be precise and consistent, such as the numbers used in financial accounting. Note that even when the literal values are equal, approximate numeric data types (**FLOAT**, **DOUBLE**, **MONETARY**) can be interpreted differently depending on the system.

CUBRID does not support the UNSIGNED type for numeric data types.

Characteristics

Precision and Scale

The precision of numeric data types is defined as the number of significant digits. This applies to both exact and approximate numeric data types.

The scale represents the number of digits following the decimal point. It is significant only in exact numeric data types. Attributes declared as exact numeric data types always have fixed precision and scale. **NUMERIC** (or **DECIMAL**) data type always has at least one-digit precision, and the scale should be between 0 and the precision declared. Scale cannot be greater than precision. For **INTEGER**, **SMALLINT**, or **BIGINT** data types, the scale is 0 (i.e. no digits following the decimal point), and the precision is fixed by the system.

Numeric Literals

Special signs can be used to input numeric values. The plus sign (+) and minus sign (-) are used to represent positive and negative numbers respectively. You can also use scientific notations. In addition, you can use currency signs specified in the system to represent currency values. The maximum precision that can be expressed by a numeric literal is 255.

Numeric Coercions

All numeric data type values can be compared with each other. To do this, automatic coercion to the common numeric data type is performed. For explicit coercion, use the **CAST** operator. When different data types are sorted or calculated in a numerical expression, the system performs automatic coercion. For example, when adding a **FLOAT** attribute value to an **INTEGER** attribute value, the system automatically coerces the **INTEGER** value to the most approximate **FLOAT** value before it performs the addition operation.

Caution Earlier version than CUBRID 2008 R2.0, the input constant value exceeds **INTEGER**, it is handled as **NUMERIC**. However, 2008 R2.0 or later versions, it is handled as **BIGINT**.

INT/INTEGER

Description

The **INTEGER** data type is used to represent integers. The value range is available is from -2,147,483,648 to +2,147,483,647. **SMALLINT** is used for small integers, and **BIGINT** is used for big integers.

```
INTEGER |
INT
```

Remark

- If a real number is entered for an **INT** type, the number is rounded to zero decimal place and the integer value is stored.
- **INTEGER** and **INT** are used interchangeably.

Example

```
If you specify 8934 as INTEGER, 8934 is stored.
If you specify 7823467 as INTEGER, 7823467 is stored.
If you specify 89.8 to an INTEGER, 90 is stored (all digits after the decimal point are rounded).
If you specify 3458901122 as INTEGER, an error occurs (if the allowable limit is exceeded).
```

SHORT/SMALLINT

Description

The **SMALLINT** data type is used to represent a small integer type. The value range is available is from -32,768 to +32,767.

```
SMALLINT |
SHORT
```

Remark

- If a real number is entered for an **SMALLINT** type, the number is rounded to zero decimal place and the integer value is stored.
- **SMALLINT** and **SHORT** are used interchangeably.

Example

```
If you specify 8934 as SMALLINT, 8934 is stored.
If you specify 34.5 as SMALLINT, 35 is stored (all digits after the decimal point are rounded).
If you specify 23467 as SMALLINT, 23467 is stored.
If you specify 89354 as SMALLINT, an error occurs (if the allowable limit is exceeded).
```

BIGINT

Description

The **BIGINT** data type is used to represent big integers. The value range is available from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

```
BIGINT
```

Remark

- If a real number is entered for a **BIG** type, the number is rounded to zero decimal place and the integer value is stored.
- Based on the precision and the range of representation, the following order applies.

  SMALLINT ⊂ INTEGER ⊂ BIGINT ⊂ NUMERIC

Example

```
If you specify 8934 as BIGINT, 8934 is stored.
If you specify 89.1 as BIGINT, 89 is stored.
If you specify 89.8 as BIGINT, 90 is stored (all digits after the decimal point are
rounded).
If you specify 3458901122 as BIGINT, 3458901122 is stored.
```

NUMERIC/DECIMAL

Description

NUMERIC or **DECIMAL** data types are used to represent fixed-point numbers. As an option, the total number of digits (precision) and the number of digits after the decimal point (scale) can be specified for definition. The minimum value for the precision *p* is 1. When the precision *p* is omitted, you cannot enter data whose integer part exceeds 15 digits because the default value is 15. If the scale s is omitted, an integer rounded to the first digit after the decimal point is returned because the default value is 0.

```
NUMERIC [(p [, s])]
```

Remark

- Precision must be equal to or greater than scale.
- Precision must be equal to or greater than the number of integer digits + scale.
- **NUMERIC**, **DECIMAL**, and **DEC** are used interchangeably.

Example

```
If you specify 12345.6789 as NUMERIC, 12346 is stored (it rounds to the first place after
the decimal point since 0 is the default value of scale).
If you specify 12345.6789 as NUMERIC(4), an error occurs (precision must be equal to or
greater than the number of integer digits).
If you declare NUMERIC(3,4), an error occurs (precision must be equal to or greater than
the scale).
If you specify 0.12345678 as NUMERIC(4,4), .1235 is stored (it rounds to the fifth place
after the decimal point).
If you specify -0.123456789 as NUMERIC(4,4), -.1235 is stored (it rounds to the fifth
place after decimal point and then prefixes a minus (-) sign).
```

FLOAT/REAL

Description

The **FLOAT** (or **REAL**) data type is used to represent floating point numbers. The value range is available from -3.402823466E+38 to -1.175494351E-38, 0, and from +1.175494351E-38 to +3.402823466E+38. It conforms to the ANSI/IEEE 754-1985 standard.

The minimum value for the precision p is 1 and the maximum value is 38. When the precision *p* is omitted or it is specified as seven or less, the data is represented as single precision (in seven significant figures) and it is converted into the **DOUBLE** data type.

```
FLOAT[(p)]
```

Remark

- **FLOAT** is in seven significant figures.
- Representable range is different based on system where CUBRID is running.
- Extra cautions are required when comparing data because the **FLOAT** type stores approximate numeric.
- **FLOAT** and **REAL** are used interchangeably.

Example

```
If you specify -1234.56789 as FLOAT, -1.234568e+003 is stored (if precision is omitted,
8th digit is rounded because it is represented as seven significant figures).
If you specify 1234.56789 as FLOAT(5), 1.234568e+003 is stored (if precision is in seven
or less, 8th digit is rounded because it is represented as seven significant figures).
If you specify 12345678.9 as FLOAT(5), 1.234568e+007 is stored (if precision is in seven
or less, 8th digit is rounded because it is represented as seven significant figures).
If you specify 12345678.9 as FLOAT(10), 1.234567890000000e+007 is stored (if precision is
in seven or greater and 38 or less, 0s are filled because it is represented as 15
significant figures).
```

DOUBLE/DOUBLE PRECISION

Description

The **DOUBLE** data type is used to represent floating point numbers. The value range is available from -1.7976931348623157E+308 to 2.2250738585072014E-308, 0, and from 2.2250738585072014E-308 to 1.7976931348623157E+308. It conforms to the ANSI/IEEE 754-1985 standard.

The precision *p* is not specified. The data specified as this data type is represented as double precision (in 15 significant figures).

```
DOUBLE
```

Remark

- **DOUBLE** is in 15 significant figures.
- Representable range is different based on system where CUBRID is running.
- Extra caution is required when comparing data because the DOUBLE type stores approximate numeric.
- **DOUBLE** and **DOUBLE PRECISION** are used interchangeably.

Example

```
If you specify 1234.56789 as DOUBLE, 1.234567890000000e+003 is stored.
```

MONETARY

Description

MONETARY data type is an approximate numeric data type. Representable range is the same as FLOAT, which is represented to two decimal places; the representable range can be different based on system. A comma is appended to every 1000th place.

```
MONETARY
```

Remark

You can use a dollar sign or a decimal point, but a comma is not allowed.

Example

```
If you specify 12345.67898934 as MONETARY, $12,345.68 is stored (it is rounded to third
decimal place).
If you specify 123456789 as MONETARY, $123,456.789.00 is stored.
```

Date/Time Types

Definition and Characteristics

Definition

DATE-TIME data types are used to represent the date or time (or both together). CUBRID supports the following data types:

Date-Time Types Supported by CUBRID

Type	Mim	Max	Note
DATE	0001-01-01	9999-12-31	0-0-0 is not allowed.
TIME	00:00:00	23:59:59	0:0:0 is not allowed.
TIMESTAMP	1970-01-01 00:00:00(GMT) 1970-01-01 09:00:00(KST)	2038-01-10 03:14:07 (GMT) 2038-01-19 12:14:07 (KST)	Note that TIMESTAMP at the point of entering data is not stored even though data is inserted into or updated in the TIMESTAMP column. Exceptionally, a value for DATETIME '0000-00-00 00:00:00' will be changed to the minimum value.
DATETIME	0001-01-01 00:00:000	9999-12-31 23:59:599	Exceptionally, a value for DATETIME '0000-00-00 00:00:00' will be changed to the minimum value.

Characteristics

Range and Resolution

- By default, the range of a time value is represented by the 24-hour system. Dates follow the Gregorian calendar. An error occurs if a value that does not meet these two constraints is entered as a date or time.
- The range of year in **DATE** is 0001 - 9999 AD.
- From the CUBRID 2008 R3.0 version,if time value is represented with two-digit numbers, a number from 00 to 69 is converted into a number from 2000 to 2069; a number from 70 to 99 is converted into a number from 1970 to 1999. In earlier than CUBRID 2008 R3.0 version, if time value is represented with two-digit numbers, a number from 01 to 99 is converted into a number from 0001 to 0099.
- The range of **TIMESTAMP** is from January 1, 1970 00:00:00 GMT to January 19, 2038 03:14:07. For KST (GMT+9), values from January 1, 1970 00:00:00 to January 19, 2038 12:14:07 can be stored.
- The results of date, time and timestamp operations may differ depending on the rounding mode. In these cases, for Time and Timestamp, the most approximate second is used as the minimum resolution; for Date, the most approximate date is used as the minimum resolution.

Coercions

The **Date-Time** types can be cast explicitly using the **CAST** operator only when they have the same field. For implicit coercion, see Implicit Type Conversion. The following table shows types that allows explicit coercions. For implicit coercion, see Arithmetic Operation and Type Casting of DATE/TIME Data Types.

Explicit Coercions

FROM TO	DATE	TIME	DATETIME	TIMESTAMP
DATE	--	X	O	O
TIME	X	--	X	X
TIMESTAMP	O	O	--	O
DATETIME	O	O	O	--

DATE

Description

The **DATE** data type is used to represent the year (yyyy), month (mm) and day (dd). Supported range is "01/01/0001" to "12/31/9999." The year can be omitted. If it is, the year value of the current system is specified automatically.

Output and input formats are as follows:

```
'mm/dd[/yyyy]'
'[yyyy-]mm-dd'
```

Remark

- All fields must be entered as integer.
- The date value is outputted in the format of 'MM/DD/YYYY' in CSQL, and it is outputted in the format of 'YYYY-MM-DD' in JDBC application programs and the CUBRID Manager.
- The **TO_DATE**() function is used to convert a character string type into a **DATE** type.

Example

```
DATE '2008-10-31' is stored as '10/31/2008'.
DATE '10/31' is stored as '10/31/2011'(if a value for year is omitted, the current year is
automatically specified).
DATE '00-10-31' is stored as '10/31/2000'.
DATE '0000-10-31' is handled as an error (a year value should be at least 1).
DATE '70-10-31' is stored as '10/31/1970'.
DATE '0070-10-31' is stored as '10/31/0070'.
```

TIME

Description

The **TIME** data type is used to represent the hour (hh), minute (mm) and second (ss). Supported range is "00:00:00" to "23:59:59." Second can be omitted; if it is, 0 seconds is specified. Both 12-hour and 24-hour notations are allowed as an input format.

The input format of **TIME** is as follows:

```
'hh:mi [:ss] [am | pm]'
```

Remark

- All items must be entered as integer.
- AM/PM time notation is used to display time in the CSQL; while the 24-hour notation is used in the CUBRID Manager.
- AM/PM can be specified in the 24-hour notation. An error occurs if the time specified does not follow the AM/PM format.

- Every time value is stored in the 24-hour notation. **db_time_decode**, one of C API functions, is used to return a value in the 24-hour notation.
- The **TO_TIME**() function is used to return a character string type into a TIME type.

Example

```
TIME '00:00:00’ is outputted as '12:00:00 AM'.
TIME '1:15' is regarded as '01:15:00 AM'.
TIME '13:15:45' is regarded as '01:15:45 PM'.
TIME '13:15:45 pm' is stored normally.
TIME '13:15:45 am' is an error (an input value does not match the AM/PM format).
```

TIMESTAMP

Description

The **TIMESTAMP** data type is used to represent a data value in which the date (year, month, date) and time (hour, minute, second) are combined. Representable range is from GMT 1970-01-01 00:00:00 to 2038-01-19 03:14:07. The **DATETIME** type can be used if the value exceeds the range or the time data in milliseconds is stored.

The input format of **TIMESTAMP** is as follows:

```
'hh:mi [:ss] [am|pm] mm/dd [/yyyy]'
'hh:mi [:ss] [am|pm] [yyyy-]mm-dd'

'mm/dd [/yyyy] hh:mi [:ss] [am|pm]'
'[yyyy-]mm-dd hh:mi [:ss] [am|pm]'
```

Remark

- All fields must be entered in integer format.
- If the year is omitted, the current year is specified by default. If the time value (hour/minute/second) is omitted, 12:00:00 AM is specified.
- You can store the timestamp value of the system in the **TIMESTAMP** type by using the **SYS_TIMESTAMP** (or **SYSTIMESTAMP**, **CURRENT_TIMESTAMP**) function. Note that the timestamp value is specified as a default value at the time of creating the table, not at the time of **INSERT** the data, if **SYS_TIMESTAMP** is specified as a **DEFAULT** value for a **TIMESTAMP** column when creating a table.
- The **TIMESTAMP()** or **TO_TIMESTAMP**() function is used to cast a character string type into a **TIMESTAMP** type.

Example

```
TIMESTAMP '10/31' is stored as '12:00:00 AM 10/31/2011'(if a value for year/time is
omitted, the default value is outputted ).
TIMESTAMP '10/31/2008' is stored as '12:00:00 AM 10/31/2008'(if a value for time is
omitted, the default value is outputted).
TIMESTAMP '13:15:45 10/31/2008' is stored as '01:15:45 PM 10/31/2008'.
TIMESTAMP '01:15:45 PM 2008-10-31' is stored as '01:15:45 PM 10/31/2008'.
TIMESTAMP '13:15:45 2008-10-31' is stored as '01:15:45 PM 10/31/2008'.
TIMESTAMP '10/31/2008 01:15:45 PM' is stored as '01:15:45 PM 10/31/2008'.
TIMESTAMP '10/31/2008 13:15:45'는 '01:15:45 PM 10/31/2008'로 출력된다.
TIMESTAMP '2008-10-31 01:15:45 PM' is stored as '01:15:45 PM 10/31/2008'.
TIMESTAMP '2008-10-31 13:15:45' is stored as '01:15:45 PM 10/31/2008'.
An error occurs on TIMESTAMP '2099-10-31 01:15:45 PM'(TIMESTAMP 표현 가능 범위 초과).
```

DATETIME

Description

The **DATETIME** data type is used to represent a data value in which the data (year, month, date) and time (hour, minute, second) are combined. Representable range is from GMT 0001-01-01 00:00:00.000 to 9999-12-31 23:59:59.999.

```
'hh:mi [:ss[.msec]] [am|pm] mm/dd [/yyyy]'
'hh:mi [:ss[.msec]] [am|pm] [yyyy-]mm-dd'
'mm/dd[/yyyy] hh:mi[:ss[.ff]] [am|pm]'
```

```
'[yyyy-]mm-dd hh:mi[:ss[.ff]] [am|pm]'
```

Remark

- All fields must be entered as integer.
- If you year is omitted, the current year is specified by default. If the value (hour, minute/second) is omitted, 12:00:00.000 AM is specified.
- You can store the timestamp value of the system in the **DATETIME** type by using the **SYS_DATETIME** (or **SYSDATETIME**, **CURRENT_DATETIME, CURRENT_DATETIME()**, **NOW()**) function. Note that the timestamp value is specified as a default value at the time of creating the table, not at the time of **INSERT** the data, if **SYS_DATETIME** is specified as a **DEFAULT** value for a **DATETIME** column when creating a table.
- The **TO_DATETIME**() function is used to cast a character string type into a **DATETIME** type.

Example

```
DATETIME '10/31' is outputted as '12:00:00.000 AM 10/31/2011' (if year/time is omitted, a
default value is outputted).
DATETIME '10/31/2008' is outputted as '12:00:00.000 AM 10/31/2008'.
DATETIME '13:15:45 10/31/2008' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '01:15:45 PM 2008-10-31' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '13:15:45 2008-10-31' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '10/31/2008 01:15:45 PM' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '10/31/2008 13:15:45' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '2008-10-31 01:15:45 PM' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '2008-10-31 13:15:45' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '2099-10-31 01:15:45 PM' is outputted as '01:15:45.000 PM 10/31/2099'.
```

Converting a String to Date/Time Type

Recommended Format for Strings in Date/Time Type

When you convert a string to Date/Time type, it is recommended to write the string in the following format:

- **DATE** Type

```
[[[[Y]Y]YY]M]MDD
[[[[Y]Y]YY]-M]M-DD
MM/DD/YYYY
```

- **TIME** Type

```
HH[:MM[:SS]] ["am"|"pm"]
```

- **DATETIME** Type

```
YYYY-MM-DD HH:MM:SS[.msec]  YY-MM-DD HH:MM:[SS[.msec]]
YY-MM-DD H
```

- **TIMESTAMP** Type

```
YYYY-MM-DD HH:MM:SS
YY-MM-DD HH:MM:[SS]
YY-MM-DD H
```

Available DATE String Format

```
[year sep] month sep day
```

- 2011-04-20
- 04-20

If a separator (*sep*) is a slash (/), strings are recognized in the following order:

```
month/day[/year]
```

- 04/20/2011
- 04/20

If you do not use a separator (*sep*), strings are recognized in the following format. It is allowed to use up to 4 digits for years and up to 2 digits for months. You must enter a 2-digit day.

```
[[[[Y]Y]YY]M]MDD
```

- 20110420
- 110420
- 420

Available TIME String Format

```
[hour]:min[:[sec]] [.[msec]] [am|pm]
```

- 09:10:15.359 am
- 09:10:15
- 09:10
- :10

```
[[[[[[Y]Y]Y]Y]M]MDD]HHMMSS[.[msec]] [am|pm]
```

- 20110420091015.359 am
- 0420091015

```
[H]HMMSS[.[msec]] [am|pm]
```

- 091015.359 am
- 91015

```
[M]MSS[.[msec]] [am|pm]
```

- 1015.359 am
- 1015

```
[S]S[.[msec]] [am|pm]
```

- 15.359 am
- 15

Note: The [H]H format was allowed in CUBRID 2008 R3.1 and the earlier versions. That is, the string '10' was converted to **TIME** '10:00:00' in the R3.1 and the earlier versions, and will be converted to **TIME** '00:00:10' in version R4.0 and later.

Available String Format in Time-Date

```
[hour]:min[:sec[.msec]] [am|pm] sep [year-]month-day
```

- 09:10:15.359 am 2011-04-20
- :10 04-20

```
[hour]:min[:sec[.msec]] [am|pm] sep month/day[/[year]]
```

- 09:10:15.359 am 04/20/2011
- :10 04/20

```
hour[:min[:sec[.[msec]]]] [am|pm] sep [year-]month-day
```

- 09:10:15.359 am 04-20
- 09 04-20

```
hour[:min[:sec[.[msec]]]] [am|pm] sep month/day[/[year]]
```

- 09:10:15.359 am 04/20
- 09 04/20

Available DATETIME String Format

```
[year sep] month sep day [sep] [sep] hour [sep min[sep sec[.[msec]]]]
```

- 04-20 09

```
month/day[/year] [sep] hour [sep min [sep sec[.[msec]]]]
```

- 04/20 09

```
year sep month sep day sep hour [sep min[sep sec[.[msec]]]]
```

- 2011-04-20 09

```
month/day/year sep hour [sep min[sep sec [.[msec]]]]
```

- 04/20/2011 09

```
YYMMDDH (??? ? ?? ?? ??? ??)
```

- 1104209

```
YYMMDDHHMM[SS[.msec]]
```

- 1104200910.359

```
YYYYMMDDHHMMSS[.msec]
```

- 201104200910.359

Rules

msec is a series of numbers representing milliseconds. The numbers after the fourth digit will be ignored.

sep represents the separator string allowed. The rules for the separator string are as follows:

- You should always use one colon (:) as a separator for the **TIME** separator.
- **DATE** and **DATETIME** strings can be represented as a series of numbers without the separator *sep*), and non-alphanumeric characters can be used as separators. The **DATETIME** string can be divided into Time and Date with a space.
- Separators do not need to be identical in the input string.
- For the Time-Date string, you can only use colon (:) for a Time separator and hyphen (-) or slash (/) for a Date separator.
- For the **DATE** string, you can use colon (:) or other separators.

The following rules will be applied to the Date part in the string.

- You can omit the year as long as the syntax allows it.
- If you enter the year as two digits, it represents the range from 1970-2069. That is, if YY<70, it is treated as 2000+YY; if YY>=70, it is treated as 1900+YY. If you enter one, three or four digit numbers for the year, the numbers will be represented as they are.
- A space before and after a string and the string next to the space are ignored. The am/pm identifier for the **DATETIME** and **TIME** strings can be recognized as part of TIME value, but are not recognized as the am/pm identifier if non-space characters are added to it.

The **TIMESTAMP** type of CUBRID consists of **DATE** type and **TIME** type, and **DATETIME** type consists of **DATE** type and **TIME** type with milliseconds being added to them. Input strings can include Date (**DATE** string), Time (**TIME** string), or both (**DATETIME** strings). You can convert a string including a specific type of data to another type, and the following rules will be applied for the conversion.

- If you convert the **DATE** string to the **DATETIME** type, the time value will be '00:00:00.'
- If you convert the **TIME** string to the **DATETIME** type, colon (:) is recognized as a date separator, so that the **TIME** string can be recognized as a date string and the time value will be '00:00:00.'
- If you convert the **DATETIME** string to the **DATE** type, the time part will be ignored from the result but the time input value format should be valid.
- You can covert the **DATETIME** string to the **TIME** type, and you must follow the following rules.
- The DATE and TIME in the string must be divided by at least one blank.
- The date part of the result value is ignored but the date input value format should be valid.
- The year in the date part must be over 4 digits (available to start with 0) or the time part must include hours and minutes ([H]H:[M]M) at least. Otherwise the date pate are recognized as the **TIME** type of the [MM]SS format, and the following string will be ignored.
- If the one of the units (year, month, date, hour, minute and second) of the **DATETIME** string is greater than 999999, it is not recognized as a number, so the string including the corresponding unit will be ignored. For example, in '2009-10-21 20:9943:10', an error occurs because the value in minutes is out of the range. However, if '2009-10-21 20:1000123:10' is entered,'2009' is recognized as the the **TIME** type of the MMSS format, so that **TIME** '00:20:09' will be returned.
- If you convert the TIME-DATE sting to the **TIME** type, the date part of the string is ignored but the date part format must be valid.
- All input strings including the time part allow [.*msec*] on conversion, but only the **DATETIME** type can be maintained. If you convert this to a type such as **DATE**, **TIMESTAMP** or **TIME**, the *msec* value is discarded.

- All conversions in the **DATETIME**, **TIME** string allow English locale following after time value or am/pm identifier written in the current locale of a server.

Example

```
SELECT CAST('420' AS DATE);

   cast('420' as date)
======================
  04/20/2011

SELECT CAST('91015' AS TIME);

   cast('91015' as time)
========================
  09:10:15 AM


SELECT CAST('110420091035.359' AS DATETIME);

   cast('110420091035.359' as datetime)
=======================================
  09:10:35.359 AM 04/20/2011

SELECT CAST('110420091035.359' AS TIMESTAMP);

   cast('110420091035.359' as timestamp)
========================================
  09:10:35 AM 04/20/2011
```

Bit Strings

Definition and Characteristics

Definition

A bit string is a sequence of bits (1's and 0's). Images (bitmaps) displayed on the computer screen can be stored as bit strings. CUBRID supports the following two types of bit strings:

- Fixed-length bit string (**BIT**)
- Variable-length bit string (**BIT VARYING**)

A bit string can be used as a method argument or an attribute domain. Bit string literals are represented in a binary or hexadecimal format. For binary format, append the string consisting of 0's and 1's to the letter **B** or append a value to the **0b** as shown example below.

```
B'1010'
0b1010
```

For hexadecimal format, append the string consisting of the numbers 0 - 9 and the letters A - F to the uppercase letter **X** or append a value to the **0x**. The following is hexadecimal representation of the same number that was represented above in binary format.

```
X'a'
0xA
```

The letters used in hexadecimal numbers are not case-sensitive. That is, X'4f' and X'4F' are considered as the same value.

Characteristics

Length

If a bit string is used in table attributes or method declarations, you must specify the maximum length. The maximum length for a bit string is 1,073,741,823 bits.

Bit String Coercion

Automatic coercion is performed between a fixed-length and a variable-length bit string for comparison. For explicit coercion, use the **CAST** operator.

BIT(n)

Description

Fixed-length binary or hexadecimal bit strings are represented as **BIT**(*n*), where *n* is the maximum number of bits. If *n* is not specified, the length is set to 1.

Remark

- *n* must be a number greater than 0.
- If the length of the string exceeds *n*, it will be processed as an error.
- If a bit string smaller than *n* is stored, the remainder of the string is filled with 0s.

Example

```
CREATE TABLE bit_tbl(a1 BIT, a2 BIT(1), a3 BIT(8), a4 BIT VARYING);
INSERT INTO bit_tbl VALUES (B'1', B'1', B'1', B'1');
INSERT INTO bit_tbl VALUES (0b1, 0b1, 0b1, 0b1);
INSERT INTO bit_tbl(a3,a4) VALUES (B'1010', B'1010');
INSERT INTO bit_tbl(a3,a4) VALUES (0xaa, 0xaa);
SELECT * FROM bit_tbl;
  a1                    a2                    a3                    a4

=========================================================================
  X'8'                  X'8'                  X'80'                 X'8'
  X'8'                  X'8'                  X'80'                 X'8'
  NULL                  NULL                  X'a0'                 X'a'
  NULL                  NULL                  X'aa'                 X'aa'
```

BIT VARYING(n)

Description

A variable-length bit string is represented as **BIT VARYING**(*n*), where *n* is the maximum number of bits. If *n* is not specified, the length is set to 1,073,741,823 (maximum value).

Remark

- If the length of the string exceeds *n*, it will be processed as an error.
- The remainder of the string is not filled with 0s even if a bit string smaller than *n* is stored.
- *n* must be a number greater than 0.

Example

```
CREATE TABLE bitvar_tbl(a1 BIT VARYING, a2 BIT VARYING(8));
INSERT INTO bitvar_tbl VALUES (B'1', B'1');
INSERT INTO bitvar_tbl VALUES (0b1010, 0b1010);
INSERT INTO bitvar_tbl VALUES (0xaa, 0xaa);
INSERT INTO bitvar_tbl(a1) VALUES (0xaaa);
SELECT * FROM bitvar_tbl;
  a1                    a2
============================================
  X'8'                  X'8'
  X'a'                  X'a'
  X'aa'                 X'aa'
  X'aaa'                NULL

INSERT INTO bitvar_tbl(a2) VALUES (0xaaa);

ERROR: Data overflow coercing X'aaa' to type bit varying.
```

Character Strings

Definition and Characteristics

Definition

CUBRID supports the following four types of character strings:

- Fixed-length character string: **CHAR**(*n*)
- Variable-length character string: **VARCHAR**(*n*)
- Fixed-length national character string: **NCHAR**(*n*)
- Variable-length national character string: **NCHAR VARYING**(*n*)

The followings are the rules that are applied when using the character string types.

- In general, single quotations are used to enclose character string. Double quotations may be used as well depending on the value of **ansi_quotes**, which is a parameter related to SQL statement. If the **ansi_quotes** value is set to **no**, character string enclosed by double quotations is handled as character string, not as an identifier. The default value is **yes**. For more information, Statement/Type-Related Parameters.
- If there are characters that can be considered to be blank (e.g. spaces, tabs, or line breaks) between two character strings, these two character strings are treated as one according to ANSI standard. For example, the following example shows that a line break exists between two character string.

```
'abc'
'def'
```

- The two strings above are considered identical to one string below.

```
'abcedf'
```

- If you want to include a single quote as part of a character string, enter two single quotes in a row. For example, the character string on the left is stored as the one on the right.

```
''abcde''fghij'              'abcde'fghij
```

- The maximum size of the token for all the character strings is 16KB.
- National character strings are used to store national (except for English alphabet) character strings in a multilingual environment. Note that **N** (uppercase) should be followed by a single quote which encloses character strings.

```
N'Ha rder'
```

Characteristics

Length

For a **CHAR** or **VARCHAR** type, specify the length (bytes) of a character string for a **NCHAR** or **NCHAR VARYING** type, specify the number of character strings (number of characters).

When the length of the character string entered exceeds the length specified, the characters in excess of the specified length are truncated.

For a fixed-length character string type such as **CHAR** or **NCHAR**, the length is fixed at the declared length. Therefore, the right part (trailing space) of the character string is filled with space characters when the string is stored. For a variable-length character string type such as **VARCHAR** or **NCHAR VARYING**, only the entered character string is stored, and the space is not filled with space characters.

The maximum length of a **CHAR** or **VARCHAR** type to be specified is 1,073,741,823 the maximum length of a **NCHAR** or **NCHAR VARYING** type to be specified is 536,870,911. The maximum length that can be input or output in a CSQL statement is 8,192 KB.

Character Set, charset

A character set (charset) is a set in which rules are defined that relate to what kind of codes can be used for encoding when specified characters (symbols) are stored in the computer.

CUBRID supports the following character sets and you can specify them as the **CUBRID_LANG** environment variable. You can store data in other character sets (e.g. utf-8), but string function or **LIKE** search are not supported.

Character Set	CUBRID_LANG
8 bits ISO 8859-1 Latin	en_US
KSC 5601-1992 (EUC_KR)	ko_KR.euckr

Any characters from the above character sets can be included in a character string (the **NULL** character is represented as '\0').

Collating Character Sets

A collation is a set of rules used for comparing characters to search or sort values stored in the database when a certain character set is specified. Therefore, such rules are applied only to character string data types such as **CHAR()** or **VARCHAR()**. For a national character string type such as **NCAHR()** or **NCHAR VARYING()**, the sorting rules are determined according to the encoding algorithm of the specified character set.

Character String Coercion

Automatic coercion takes place between a fixed-length and a variable-length character string for the comparison of two characters, applicable only to characters that belong to the same character set. For example, when you extract a column value from a CHAR(5) data type and insert it into a column with a CHAR(10) data type, the data type is automatically coerced to CHAR(10). If you want to coerce a character string explicitly, use the **CAST** operator (See CAST Operator).

CHAR(n)

Description

A fixed-length character string is represented as **CHAR***(n)*, in which n is the number of bytes in an ASCII character string. For the English alphabet, each character takes up one byte. However, for Korean characters, note that the number of bytes taken up by each character differs depending on the character set of the data input environment (e.g. EUC-KR: 2 bytes, utf-8: 3 bytes). If *n* is not specified, the length is set to the default value 1.

When the length of a character string exceeds *n*, they are truncated. When character string which is shorter than *n* is stored, whitespace characters are used to fill up the trailing space.

CHAR(*n*) and **CHARACTER**(*n*) are used interchangeably.

Remark

- The **CHAR** data type is always based on the ISO 8859-1 (Latin-1) character set.
- *n* is an integer between 1 and 1,073,741,823 (1G).
- Empty quotes (' ') are used to represent a blank string. In this case, the return value of the **LENGTH** function is not 0, but is the fixed length defined in **CHAR**(*n*). That is, if you enter a blank string into a column with **CHAR**(10), the **LENGTH** is 10; if you enter a blank value into a **CHAR** with no length specified, the **LENGTH** is the default value 1.
- Space characters used as filling characters are considered to be smaller than any other characters, including special characters.

Example 1

```
If you specify 'pacesetter' as CHAR(12), 'pacesetter ' is stored (a 10-character string
plus two whitespace characters).
If you specify 'pacesetter ' as CHAR(10), 'pacesetter' is stored (a 10-character string;
two whitespace characters are truncated).
If you specify 'pacesetter' as CHAR(4), 'pace' is stored (truncated as the length of the
character string is greater than 4).
If you specify 'p ' as CHAR, 'p' is stored (if n is not specified, the length is set to
the default value 1).
```

Example 2

```
If you specify '큐브리드' as CHAR(10) in the EUC-KR encoding, it is processed normally.
```

```
If you specify '큐브리드' as CHAR(10) and the use the CHAR_LENGTH() function in the EUC-KR
encoding, 10 is stored.
If you specify '큐브리드, as CHAR(10) in the utf-8 encoding, the last character is broken
(one Korean character takes up three bytes in the utf-8 encoding so it requires two more
bytes).
If you specify '큐브리드' as CHAR(12) in the utf-8 encoding, it is processed normally.
```

VARCHAR(n)/CHAR VARYING(n)

Description

Variable-length character strings are represented as **VARCHAR**(*n*), where *n* is the maximum number of ASCII character strings. Each English character takes up one byte. For Korean characters, note that the number of bytes taken up by each character differs depending on the character set of the data input environment (e.g. EUC-KR: 2 bytes, utf-8: 3 bytes). If *n* is not specified, the length is set to the maximum length of 1,073,741,823.

When the length of a character string exceeds *n,* they are truncated. When character string which is shorter than *n* is stored, whitespace characters are used to fill up the trailing space; for **VARCHAR**(*n*), the length of string used are stored.

VARCHAR(*n*), **CHARACTER, VARYING**(*n*), and **CHAR VARYING**(*n*) are used interchangeably.

Remark

- **STRING** is the same as the **VARCHAR** (maximum length).
- *n* is an integer between 1 and 1,073,741,823 (1G).
- Empty quotes (' ') are used to represent a blank string. In this case, the return value of the **LENGTH** function is not 0.

Example 1

```
If you specify 'pacesetter' as CHAR(4), 'pace' is stored (truncated as the length of the
character string is greater than 4).
If you specify 'pacesetter' as VARCHAR(12), 'pacesetter' is stored (a 10-character string).
If you specify 'pacesetter ' as VARCHAR(12), 'pacesetter ' is stored (a 10-character
string plus two whitespace characters).
If you specify 'pacesetter ' as VARCHAR(10), 'pacesetter' is stored (a 10-character string;
two whitespace characters are truncated).
If you specify 'p ' as VARCHAR, 'p' is stored (if n is not specified, the default value
1,073,741,823 is used, and the trailing space is not filled with whitespace characters).
```

Example 2

```
If you specify '큐브리드' as VARCHAR(10) in the EUC-KR encoding, it is processed normally.
If you specify '큐브리드' as CHAR(10) and then use CHAR_LENGTH() function in the EUC-KR
encoding, 8 is stored.
If you specify '큐브리드, as CHAR(10) in the utf-8 encoding, the last character is broken
(one Korean character takes up three bytes in the utf-8 encoding so it requires two more
bytes).
If you specify '큐브리드' as VARCHAR(12) in the utf-8 encoding, it is processed normally.
```

STRING

Description

STRING is a variable-length character string data type. **STRING** is the same as the VARCHAR with the length specified to the maximum value. That is, **STRING** and **VARCHAR**(1,073,741,823) have the same value.

NCHAR(n)

Description

NCHAR(*n*) is used to store non-English character strings. It can be used only for character sets supported by CUBRID described above. n is the number of characters. If *n* is omitted, the length is specified as the default value 1. When the length of a character string exceeds *n*, they are truncated. When character string which is shorter than *n* is stored, whitespace characters are used to fill up the space.

To store a Korean character string as a national character string type, you must set the locale of the operating system to Korean, or set the value of the **CUBRID_LANG** environment variable to **ko_KR.euckr** before creating the table.

Remark

- *n* is an integer between 1 and 5,368,709,111.
- The number of national character sets that can be used in a single database is set to be one. For example, 8-bit ISO 8889-1 (Latin-1) and EUC code sets cannot be used simultaneously in the same database.
- An error occurs if a non-national character string (whether it is fixed-length or variable-length) is specified for an attribute declared as a national character string.
- Using two different character code sets at the same time also causes an error.

Example

```
If you specify ‘큐브리드’ as NCHAR(5) in the EUC-KR encoding, it is processed normally.
If you specify '큐브리드' as NCHAR(5) and then use the CHAR_LENGTH() function in the EUC-KR
encoding, 5 is stored.
If you specify '큐브리드' as NCHAR(5) in the utf-8 encoding, an error occurs (utf-8
character set is not supported).
```

NCHAR VARYING(n)

Description

NCHAR VARYING(*n*) is a variable-length character string type. For details, see description and note of NCHAR(n). The difference is that the right part (trailing space) of the character string is not filled with whitespace characters, even when the number of strings is smaller than n.

NCHAR VARYING(*n*), **NATIONAL CHAR VARYING**(*n*), and **NATIONAL CHARACTER VARYING(n)** are used interchangeably.

Example

```
If you specify '큐브리드' as NCHAR VARYING(5) in the EUC-KR encoding, it is processed
normally.
If you specify '큐브리드' as NCHAR VARYING(5) and then use CHAR_LENGTH() function in the
EUC-KR encoding, 4 is stored.
If you specify '큐브리드' as HCHAR VARYING(5) in the utf-8 encoding, an error occurs (utf-8
character set is not supported).
```

Special Character Escape

Description

CUBRID supports two kinds of methods to escape special characters. One is using quotes and the other is using backslash (\).

Escape with Quotes

If you set yes for the system parameter **ansi_quotes** in the **cubrid.conf** file, you can use both double quotes (") and singe quotes (') to wrap strings. The default value for the **ansi_quotes** parameter is **no**, and you can use only single quotes to wrap the string. The numbers 2 and 3 below are applied only if you set for the **ansi_quotes** parameter to yes.

- You should use two single quotes ('') for the single quotes included in the strings wrapped in single quotes.
- You should use two double quotes ("") for the double quotes included in the strings wrapped in double quotes.
- You don't need to escape the single quotes included in the string wrapped in double quotes.
- You don't need to escape the double quotes included in the string wrapped in single quotes.

Escape with Backslash

You can use escape using backslash (\) only if you set no for the system parameter **no_backslash_escapes** in the **cubrid.conf** file. The default value for the **no_backslash_escapes** parameter is **yes**. Depending on the input value, the following are the special characters.

- \' : Single quotes (')
- \" : Double quotes (")
- \n : Newline, linefeed character
- \r : Carriage return character
- \t : Tab character
- \\ : Backslash
- \% : Percent sign (%). For more information, see the following description.
- _ : Underbar (_). For more information, see the following description.

For all other escapes, the backslash will be ignored. For example, "\x" is the same as entering only "x".

\% and **_** are used in the pattern matching syntax such as **LIKE** to search percent signs and underbars and are used as a wildcard character if there is no backslash. Outside of the pattern matching syntax, "\%"and "_" are recognized as normal strings not wildcard characters. For more information, see LIKE Predicate.

Example 1

The following is the result of executing Escape if a value for the system parameter **ansi_quotes** in the **cubrid.conf** file is no, and a value for **no_backslash_escapes** is no.

```
SELECT STRCMP('single quotes test('')', 'single quotes test(\')');

   strcmp('single quotes test('')', 'single quotes test('')')
=============================================================
                                                            0

SELECT STRCMP("\a\b\c\d\e\f\g\h\i\j\k\l\m\n\o\p\q\r\s\t\u\v\w\x\y\z",
"a\bcdefghijklm\nopq\rs\tuvwxyz");

   strcmp('abcdefghijklm
s       uvwxyz', 'abcdefghijklm
s       uvwxyz')
=====================================================================
                                                                    0

SELECT LENGTH('\\');

   char_length('\')
===================
                  1
```

Example 2

The following is the result of executing Escape if a value for the system parameter **ansi_quotes** in the **cubrid.conf** file is yes, and a value for **no_backslash_escapes** is yes.

```
SELECT STRCMP('single quotes test('')', 'single quotes test(\')');
```

```
In the command from line 2,
ERROR: unterminated string

In the command from line 2,
ERROR: syntax error, unexpected UNTERMINATED_STRING


SELECT STRCMP("\a\b\c\d\e\f\g\h\i\j\k\l\m\n\o\p\q\r\s\t\u\v\w\x\y\z",
"a\bcdefghijklm\nopq\rs\tuvwxyz");

In line 1, column 18,
ERROR: [\a\b\c\d\e\f\g\h\i\j\k\l\m\n\o\p\q\r\s\t\u\v\w\x\y\z] is not defined.

In line 1, column 18,
ERROR: [a\bcdefghijklm\nopq\rs\tuvwxyz] is not defined.

SELECT LENGTH('\\');

   char_length('\\')
====================
                   2
```

Example 3

The following is the result of executing Escape if a value for the system parameter **ansi_quotes** in the **cubrid.conf** file is yes, and a value for **no_backslash_escapes** is no.

```
CREATE TABLE t1 (a varchar(200));
INSERT INTO t1 VALUES ('aaabbb'), ('aaa%');

SELECT a FROM t1 WHERE a LIKE 'aaa\%' escape '\\';

  a
======================
  'aaa%'
```

BLOB/CLOB Data Types

Definition and Characteristics

Definition

An External **LOB** type is data to process Large Object, such as text or images. When LOB-type data is created and inserted, it will be stored in a file to an external storage, and the location information of the relevant file (**LOB** Locator) will be stored in the CUBRID database. If the **LOB** Locator is deleted from the database, the relevant file that was stored in the external storage will be deleted as well. CUBRID supports the following two types of **LOB**:

- Binary Large Object (**BLOB**)
- Character Large Object (**CLOB**)

Related Terms

- **LOB** (Large Object) : Large-sized objects such as binaries or text.
- **FBO** (File Based Object) : An object that stores data of the database in an external file.
- **External LOB** : An object better known as FBO, which stores **LOB** data in a file into an external DB. It is supported by CUBRID. Internal **LOB** is an object that stores **LOB** data inside the DB.
- **External Storage** : An external storage to store LOB (example : POSIX file system).
- **LOB Locator** : The path name of a file stored in external storage.
- **LOB Data** : Details of a file in a specific location of LOB Locator.

File Name

When storing LOB data in external storage, the following naming convention will be applied:

```
{table_name}_{unique_name}
```

- *table_name* : It is inserted as a prefix and able to store the **LOB** data of many tables in one external storage.
- *unique_name* : The random name created by the DB server.

Default Storage

- **LOB** data is stored in the local file system of the DB server. LOB data is stored in the path specified in the **-lob-base-path option** value of **cubrid createdb**; if this value is omitted, the data will be stored in the [db-vol path]/lob path where the database volume will be created. For more details, see Database Creation and Storage Creation and Management.
- If the relevant path is deleted despite a **LOB** data file path being registered in the database location file (**databases.txt**), please note that the utility that operates in database server (cub_server) and standalone will not function normally.

BLOC/CLOB

BLOB

- A type that stores binary data outside the database.
- The maximum length of **BLOB** data is the maximum file size creatable in an external storage.
- In SQL statements, the **BLOB** type expresses the input and output value in a bit array. That is, it is compatible with the **BIT**(n) and **BIT VARYING**(n) types, and only an explicit type change is allowed. If data lengths differ from one another, the maximum length is truncated to fit the smaller one.
- When converting the **BLOB** type value to a binary value, the length of the converted data cannot exceed 1GB. When converting binary data to the **BLOB** type, the size of the converted data cannot exceed the maximum file size provided by the **BLOB** storage.

CLOB

- A type that stores character string data outside the database.
- The maximum length of **CLOB** data is the maximum file size creatable in an external storage.
- In SQL statements, the CLOB type expresses the input and output value in a character string. That is, it is compatible with the **CHAR**(n), **VARCHAR**(n), **NCHAR**(n), and **NCHAR VARYING**(n) types. However, only an explicit type change is allowed, and if data lengths are different from one another, the maximum length is truncated to fit to the smaller one.
- When converting the **CLOB** type value to a character string, the length of the converted data cannot exceed 1 GB. When converting a character string to the **CLOB** type, the size of the converted data cannot exceed the maximum file size provided by the **CLOB** storage.

Creating and Altering Columns

Description

BLOB/**CLOB** type columns can be created/added/deleted by using a **CREATE TABLE** statement or an **ALTER TABLE** statement.

Note

- You cannot create the index file for a **LOB** type column.
- You cannot define the **PRIMARY KEY**, **FOREIGN KEY**, **UNIQUE**, and **NOT NULL** constraints for a **LOB** type column. However, **SHARED** property cannot be defined and **DEFAULT** property can only be defined by the **NULL** value.
- **LOB** type column/data cannot be the element of collection type data.
- If you are deleting a record containing a **LOB** type column, all files located inside a **LOB** column value (Locator) and the external storage will be deleted. When a record containing a LOB type column is deleted in a basic key table, and a record of a foreign key table that refers to the foregoing details is deleted at the same time, all **LOB** files located in a **LOB** column value (Locator) and the external storage will be deleted. However, if the relevant table is deleted by using a **DROP TABLE** statement, or a **LOB** column is deleted by using an **ALTER TABLE...DROP** statement, only a **LOB** column value (**LOB** Locator) is deleted, and the **LOB** files inside the external storage which a **LOB** column refers to will not be deleted.

Example

```
-- creating a table and CLOB column
CREATE TABLE doc_t (doc_id VARCHAR(64) PRIMARY KEY, content CLOB);

-- an error occurs when UNIQUE constraint is defined on CLOB column
ALTER TABLE doc_t ADD CONSTRAINT content_unique UNIQUE(content);

-- an error occurs when creating an index on CLOB column
CREATE INDEX ON doc_t (content);

-- creating a table and BLOB column
CREATE TABLE image_t (image_id VARCHAR(36) PRIMARY KEY, doc_id VARCHAR(64) NOT NULL, image
BLOB);

-- an error occurs when adding a BOLB column with NOT NULL constraint
ALTER TABLE image_t ADD COLUMN thumbnail BLOB NOT NULL;

-- an error occurs when adding a BLOB column with DEFAULT attribute
ALTER TABLE image_t ADD COLUMN thumbnail2 BLOB DEFAULT BIT_TO_BLOB(X'010101');
```

Storing and Updating Columns

Description

In a **BLOB**/**CLOB** type column, each **BLOB**/**CLOB** type value is stored, and if binary or character string data is input, you must explicitly change the types by using each **BIT_TO_BLOB**( )/**CHAR_TO_CLOB**( ) function.

If a value is input in a **LOB** column by using an **INSERT** statement, a file is created in an external storage internally and the relevant data is stored; the relevant file path (Locator) is stored in an actual column value.

If a record containing a **LOB** column uses a **DELETE** statement, a file to which the relevant **LOB** column refers will be deleted simultaneously. If a **LOB** column value is changed using an **UPDATE** statement, the column value will be changed following the operation below, according to whether a new value is **NULL** or not.

- If a **LOB** type column value is changed to a value that is not **NULL** : If a Locator that refers to an external file is already available in a **LOB** column, the relevant file will be deleted. A new file is created afterwards. After storing a value that is not **NULL**, a Locator for a new file will be stored in a **LOB** column value.
- If changing a **LOB** type column value to **NULL** : If a Locator that refers to an external file is already available in a **LOB** column, the relevant file will be deleted. And then **NULL** is stored in a **LOB** column value.

Example

```
-- inserting data after explicit type conversion into CLOB type column
INSERT INTO doc_t (doc_id, content) VALUES ('doc-1', CHAR_TO_CLOB('This is a Dog'));
INSERT INTO doc_t (doc_id, content) VALUES ('doc-2', CHAR_TO_CLOB('This is a Cat'));

-- inserting data after explicit type conversion into BLOB type column
INSERT INTO image_t VALUES ('image-0', 'doc-0', BIT_TO_BLOB(X'000001'));
INSERT INTO image_t VALUES ('image-1', 'doc-1', BIT_TO_BLOB(X'000010'));
INSERT INTO image_t VALUES ('image-2', 'doc-2', BIT_TO_BLOB(X'000100'));

-- inserting data from a sub-query result
INSERT INTO image_t SELECT 'image-1010', 'doc-1010', image FROM image_t WHERE image_id =
'image-0';

-- updating CLOB column value to NULL
UPDATE doc_t SET content = NULL WHERE doc_id = 'doc-1';

-- updating CLOB column value
UPDATE doc_t SET content = CHAR_TO_CLOB('This is a Dog') WHERE doc_id = 'doc-1';

-- updating BLOB column value
UPDATE image_t SET image = (SELECT image FROM image_t WHERE image_id = 'image-0') WHERE
image_id = 'image-1';

-- deleting BLOB column value and its referencing files
DELETE FROM image_t WHERE image_id = 'image-1010';
```

Getting Column Values

Description

When you get a **LOB** type column, the data stored in a file to which the column refers will be displayed. You can execute an explicit type change by using **CAST** operator, **CLOB_TO_CHAR**( ) function, and **BLOB_TO_BIT**( ) function.

Note

- If the query is executed in CSQL, a column value (Locator) will be displayed, instead of the data stored in a file. To display the data to which a **BLOB**/**CLOB** column refers, it must be changed to strings by using **CLOB_TO_CHAR**( ) function.
- To use the string process function, the strings need to be converted by using the **CLOB_TO_CHAR**( ) function.
- You cannot specify a **LOB** column in **GROUP BY** clause and **ORDER BY** clause.
- Comparison operators, relational operators, **IN**, **NOT IN** operators cannot be used to compare **LOB** columns. However, **IS NULL** expression can be used to compare whether it is a **LOB** column value (Locator) or **NULL**. This means that **TRUE** will be returned when a column value is **NULL**, and if a column value is **NULL**, there is no file to store **LOB** data.
- When a **LOB** column is created, and the file is deleted after data input, a **LOB** column value (Locator) will become a state that is referring to an invalid file. As such, using **CLOB_TO_CHAR**( ), **BLOB_TO_BIT**( ), **CLOB_LENGTH**( ), and **BLOB_LENGTH**( ) functions on the columns that have mismatching **LOB** Locator and a **LOB** data file enables them to display **NULL**.

Example

```
-- displaying locator value when selecting CLOB and BLOB column in CSQL interpreter
SELECT doc_t.doc_id, content, image FROM doc_t, image_t WHERE doc_t.doc_id =
image_t.doc_id;

  doc_id                content               image
==================================================================
  'doc-1'               file:/home1/data1/ces_658/doc_t.00001282208855807171_7329  file:/
home1/data1/ces_318/image_t.00001282208855809474_7474
  'doc-2'               file:/home1/data1/ces_180/doc_t.00001282208854194135_5598  file:/
home1/data1/ces_519/image_t.00001282208854205773_1215

2 rows selected.

-- using string functions after coercing its type by CLOB_TO_CHAR( )
SELECT CLOB_TO_CHAR(content), SUBSTRING(CLOB_TO_CHAR(content), 10) FROM doc_t;

   clob_to_char(content)  substring( clob_to_char(content) from 10)
============================================
  'This is a Dog'       ' Dog'
  'This is a Cat'       ' Cat'

2 rows selected.

SELECT CLOB_TO_CHAR(content) FROM doc_t WHERE CLOB_TO_CHAR(content) LIKE '%Dog%';

   clob_to_char(content)
======================
  'This is a Dog'

SELECT CLOB_TO_CHAR(content) FROM doc_t ORDER BY CLOB_TO_CHAR(content)

   clob_to_char(content)
======================
  'This is a Cat'
  'This is a Dog'

-- an error occurs when LOB column specified in WHERE/ORDER BY/GROUP BY clauses
SELECT * FROM doc_t WHERE content LIKE 'This%';
SELECT * FROM doc_t ORDER BY content;
```

Functions and Operators

CAST Operator

By using **CAST** operator, you can execute an explicit type change between **BLOB**/**CLOB** type and binary type/string type. For more details, see CAST Operator.

Syntax

```
CAST (<bit_type_column_or_value> AS CLOB)
CAST (<bit_type_column_or_value> AS BLOB)
CAST (<char_type_column_or_value> AS BLOB)
CAST (<char_type_column_or_value> AS CLOB)
```

LOB Data Process and Type Change Functions

The next table shows the functions provided to process and change BLOB/CLOB types.

Function Expression	Description
CLOB_TO_CHAR (<*clob_type_column*>)	Changes number type, date/time type, and **CLOB** type to **VARCHA**R type.
BLOB_TO_BIT (<*blob_type_column*>)	Changes **BLOB** type to **VARYING BIT** type.
CHAR_TO_CLOB(<*char_type_column_or_value*>)	Changes text string type (**CHAR**, **VARCHAR**, **NCHAR**, **NVACHAR**) to **CLOB** type.
BIT_TO_BLOB(<*blob_type_column_or_value*>)	Changes bit array type (**BIT**, **VARYING BIT**) to **BLOB** type.
CHAR_TO_BLOB(<*char_type_colulmn_or_value* >)	Changes text string type (**CHAR**, **VARCHAR**, **NCHAR**, **NVACHAR**) to **BLOB** type.
CLOB_FROM_FILE(<*file_pathname*>)	Reads file details from the file path of **VARCHAR** type and changes to **CLOB** type data. <*file_pathname*> is analyzed to a path of server which is operated by the DB client, such as CAS or CSQL. If a path is specified targeting this, the upper path will be the current work direction of the process. The statement that calls this function will not cache execution plans.
BLOB_FROM_FILE(<*file_pathname*>)	Reads file details from the file path of **VARCHAR** type, and changes to BLOB type data. The file path specified in is interpreted using the same method as the **CLOB_FROM_FILE**( ) function.
CLOB_LENGTH(<*clob_column*>)	Returns the length of LOB data stored in a **CLOB** file in bytes.
BLOB_LENGTH(<*blob_column*>)	Returns the length of LOB data stored in a **BLOB** file in bytes.
<*blob_or_clob_column*> **IS NULL**	Use an **IS NULL** expression to compare whether it is a **LOB** column value (Locator) or **NULL**; returns **TRUE** if **NULL**.

Creating and Managing Storage

LOB File Path Specification

By default, the **LOB** data file is stored in the <db-volumn-path>/lob directory where database volume is created. However, if the **--lob-base-path** option of **cubrid createdb** utility is used when creating the database, a **LOB** data file can be stored in the directory specified by option value. However, if there is no directory specified by option value, attempt to create a directory, and display an error message if it fails to create the directory. For more details, see the **--lob-base-path** option in Creating Database.

```
# image_db volume is created in the current work directory, and a LOB data file will be
stored.
cubrid createdb image_db

# LOB data file is stored in the "/home1/data1" path within a local file system.
```

```
cubrid createdb --lob-base-path="file:/home1/data1" image_db
```

Check LOB File Store Directory

```
# You can check a directory where a LOB file will be stored by executing the cubrid
spacedb utility.
cubrid spacedb image_db

Space description for database 'image_db' with pagesize 16.0K. (log pagesize: 16.0K)

Volid  Purpose  total_size  free_size  Vol Name

    0  GENERIC      512.0M     510.1M  /home1/data1/image_db
Space description for temporary volumes for database 'image_db' with pagesize 16.0K.

Volid  Purpose  total_size  free_size  Vol Name

LOB space description file:/home1/data1
```

Change or Expand LOB File Store Directory

Secure disk space to create additional file storage, expand the **lob-base-path** of **databases.txt**, and change to the disk location. Restart the database server to apply the changes made to **databases.txt**. However, even if you change the **lob-base-path** of **databases.txt**, access to the **LOB** data saved in a previous storage is possible.

```
# You can change to a new directory from the lob-base-path of databases.txt file.
sh> cat $CUBRID_DATABASES/databases.txt
#db-name         vol-path            db-host        log-path         lob-base-path
image_db         /home1/data1        localhost      /home1/data1     file:/home1/data2
```

Backup and Recovery of LOB Files

While backup/recovery is not supported for **LOB** data files, **LOB** type column value (Locator) is supported with such service.

Copying a Database with LOB Files

If you are copying a database by using the **cubrid copydb** utility, you must configure the **databases.txt** additionally, as the **LOB** file directory path will not be copied if the related option is not specified. For more details, see the **-B** and **--copy-lob-path** options in Copying/Moving Database.

Supporting and Recovering Transactions

Definition

Commit/rollback for **LOB** data changes are supported. That is, it ensures the validation of mapping between **LOB** Locator and actual **LOB** data within transactions, and it supports recovery during DB errors. This means that an error will be displayed in case of mapping errors between **LOB** Locator and **LOB** data due to the rollback of the relevant transactions, as the database is terminated during transactions. See the example below.

Example

```
;AUTOCOMMIT OFF

CREATE TABLE doc_t (doc_id VARCHAR(64) PRIMARY KEY, content CLOB);
INSERT INTO doc_t VALUES ('doc-10', CHAR_TO_CLOB('This is content'));
COMMIT;
UPDATE doc_t SET content = CHAR_TO_CLOB('This is content 2') where doc_id = 'doc-10';
ROLLBACK;
SELECT doc_id, CLOB_TO_CHAR(content) FROM doc_t WHERE doc_id = 'doc-10';
  doc_id   content
=========================================================
  'doc-10'  'This is content '

INSERT INTO doc_t VALUES ('doc-11', CHAR_TO_CLOB ('This is content'));
COMMIT;
UPDATE doc_t SET content = CHAR_TO_CLOB('This is content 3') WHERE doc_id = 'doc-11';
```

```
-- system crash occurred and then restart server
SELECT doc_id, CLOB_TO_CHAR(content) FROM doc_t WHERE doc_id = 'doc-11';

-- Error : LOB Locator references to the previous LOB data because only LOB Locator is
rollbacked.
```

Note

- When selecting **LOB** data in an application through a driver such as JDBC, the driver can get ResultSet from DB server and fetch the record while changing the cursor location on Resultset. That is, only Locator, the **LOB** column value, is stored at the time ResultSet is imported, and **LOB** data that is referred by a File Locator will be fetched from the file Locator at the time a record is fetched. Therefore, if **LOB** data is updated between two different points of time, there could be an error, as the mapping of **LOB** Locator and actual **LOB** data will be invalid.
- Since backup/recovery is supported only for **LOB** type column value (Locator), an error is likely to occur, as the mapping of **LOB** Locator and LOB data is invalid if recovery is performed based on a specific point of time.
- If the DB is operated in different equipment like S1 and S2, and you want to store **LOB** data in the DB of S1 equipment to S2 equipment, you must read the **LOB** data which the **LOB** column value of S1 equipment is referring to, and **INSERT LOB**. The **LOB** column value (Locator) of S1 equipment is valid only in the relevant local system.

Caution Up to CUBRID 2008 R3.0, Large Objects are processed by using **glo** (Generalized Large Object) classes. However, the **glo** classes has been deprecated since the CUBRID 2008 R3.1. Instead of it, **LOB**/**CLOB** data type is supported. Hence, both DB schema and application must be modified when upgrading CUBRID in an environment using the previous version of **glo** classes.

Collection Types

Definition and Characteristics

Definition

Allowing multiple data values to be stored in a single attribute is an extended feature of relational database. Elements of a collection are possible to have different domain each other. The domain can be one of the primitive data types or classes excluding virtual classes. For example, **SET** (INTEGER, tbl_1) can specify an integer or a set of row values of the user-defined class tbl_1 as a domain. When a domain list is not specified (e.g. **SET** ( )), all data types are allowed as elements including user-defined classes.

The data of a collection-type column with at least two domain lists can be retrieved by using the **csql** utility or the C-API. It cannot be retrieved in CUBRID manager or CUBRID API (JDBC, ODBC, OLEDB, PHP, CCI).

Collection Types Supported by CUBRID

Type	Description	Definition	Input Data	Stored Data
SET	A union which does not allow duplicates	col_name SET VARCHAR(20) col_name SET (int, VARCHAR(20))	{'c','c','c','b','b', 'a'}{3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}	{'a','b','c'} {0,1,2,3,'a','b','c'}
MULTISET	A union which allows duplicates	col_name MULTISET VARCHAR(20) col_name MULTISET (int, VARCHAR(20))	{'c','c','c','b','b', 'a'}{3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}	{'a','b','b','c','c','c'} {0,1,2,2,3,3,3,'a','b','b', 'c','c','c'}
LIST SEQUENCE SEQUENCE	A union which allows duplicates and stores data in the	col_name LIST VARCHAR(20) col_name LIST (int, VARCHAR(20))	{'c','c','c','b','b', 'a'} {3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}	{'c','c','c','b','b','a'} {3,3,3,2,2,1,0,'c','c','c','b','b','a'}

	order of input	

As you see the table above, the value specified as a collection type can be inputted with braces ('{', '}') each value is separated with a comma (,).

Characteristics

Coercions

If the specified domains are identical, the collection types can be cast explicitly by using the **CAST** operator. The following table shows the collection types that allow explicit coercions.

Explicit Coercions

		TO		
		SET	**MULTISET**	**LIST**
FROM	**SET**	-	O	O
	MULTISET	O	-	X
	LIST	O	O	-

SET

Description

SET is a set type in which each element has different values. Elements of a **SET** can have many different data types or even instances of different classes.

Example

```
CREATE TABLE set_tbl ( col_1 set(int, CHAR(1)));
INSERT INTO set_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b','a'});
INSERT INTO set_tbl VALUES ({NULL});
INSERT INTO set_tbl VALUES ({''});
SELECT * FROM set_tbl;
  col_1
======================
{0, 1, 2, 3, 'a', 'b', 'c'}
{NULL}
{' '}

SELECT CAST(col_1 AS MULTISET), CAST(col_1 AS LIST) FROM set_tbl;
   cast(col_1 as multiset)   cast(col_1 as sequence)
============================================
  {0, 1, 2, 3, 'a', 'b', 'c'}  {0, 1, 2, 3, 'a', 'b', 'c'}
  {NULL}  {NULL}
  {' '}  {' '}

INSERT INTO set_tbl VALUES ('');

ERROR: Cannot coerce '' to type set.
```

MULTISET

Description

MULTISET is a collection type in which duplicated elements are allowed. Elements of a **MULTISET** can have many different data types or even instances of different classes.

Example

```
CREATE TABLE multiset_tbl ( col_1 multiset(int, CHAR(1)));
INSERT INTO multiset_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b', 'a'});
SELECT * FROM multiset_tbl;
  col_1
======================
  {0, 1, 2, 2, 3, 3, 3, 'a', 'b', 'b', 'c', 'c', 'c'}

SELECT CAST(col_1 AS SET), CAST(col_1 AS LIST) FROM multiset_tbl;
   cast(col_1 as set)   cast(col_1 as sequence)
============================================
  {0, 1, 2, 3, 'a', 'b', 'c'}  {3, 3, 3, 2, 2, 1, 0, 'c', 'c', 'c', 'b', 'b', 'a
'}
```

LIST/SEQUENCE

Description

LIST (=**SEQUENCE**) is a collection type in which the input order of elements is preserved, and duplications are allowed. Elements of a **LIST** can have many different data types or even instances of different classes.

Example

```
CREATE TABLE list_tbl ( col_1 list(int, CHAR(1)));
INSERT INTO list_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b', 'a'});
SELECT * FROM list_tbl;
  col_1
======================
{3, 3, 3, 2, 2, 1, 0, 'c', 'c', 'c', 'b', 'b', 'a'}

SELECT CAST(col_1 AS SET), CAST(col_1 AS MULTISET) FROM list_tbl;
   cast(col_1 as set)  cast(col_1 as multiset)
============================================
  {0, 1, 2, 3, 'a', 'b', 'c'}  {0, 1, 2, 2, 3, 3, 3, 'a', 'b', 'b', 'c', 'c', 'c
'}
```

Implicit Type Conversion

Rules

An implicit type conversion represents an automatic conversion of a type of expression to a corresponding type. **SET**, **MULTISET**, **LIST** and **SEQUENCE** should be converted explicitly.

If you convert the **DATETIME** and **TIMESTAMP** types to the **DATE** type or **TIME** type, data loss may occur. If you convert the **DATE** type to the **DATETIME** type or **TIMESTAMP** type, the time will be set to '12:00:00: AM.'

If you convert a string type or an exact numeric type to a floating-point numeric type, it may not be accurate. Because a string type and an exact type use a decimal precision to represent the value, but a floating-point numeric type uses a binary precision.

The implicit type conversion executed by CUBRID is as follows:

Implicit Type Conversion Table 1

From \ To	DATETIME	DATE	TIME	TIMESTAMP	DOUBLE	FLOAT	NUMERIC	BIGINT
DATETIME	-	O	O	O				
DATE	O	-		O				
TIME			-					
TIMESTAMP	O	O	O	-				
DOUBLE					-	O	O	O
FLOAT					O	-	O	O

NUMERIC					O	O	-	O
BIGINT					O	O	O	-
INT					O	O	O	O
SHORT					O	O	O	O
MONETARY					O	O	O	O
BIT								
VARBIT								
CHAR	O	O	O	O	O	O	O	O
VARCHAR	O	O	O	O	O	O	O	O
NCHAR	O	O	O	O	O	O	O	O
VARNCHAR	O	O	O	O	O	O	O	O

Implicit Type Conversion Table 2

From \ To	INT	SHORT	MONETARY	BIT	VARBIT	CHAR	VARCHAR	NCHAR	VARNCHAR
DATETIME						O	O	O	O
DATE						O	O	O	O
TIME						O	O	O	O
TIMESTAMP						O	O	O	O
DOUBLE	O	O	O			O	O	O	O
FLOAT	O	O	O			O	O	O	O
NUMERIC	O	O	O			O	O	O	O
BIGINT	O	O	O			O	O	O	O
INT	-	O	O			O	O	O	O
SHORT	O	-	O			O	O	O	O
MONETARY	O	O	-			O	O	O	O
BIT				-	O	O	O	O	O
VARBIT				O	-	O	O	O	O
CHAR	O	O	O	O	O	-	O	O	O
VARCHAR	O	O	O	O	O	O	-	O	O
NCHAR	O	O	O	O	O	O	O	-	O
VARNCHAR	O	O	O	O	O	O	O	O	-

INSERT and UPDATE

The type will be converted to the type of the column affected.

```
CREATE TABLE t(i INT);
INSERT INTO t VALUES('123');

SELECT * FROM t;

            i
=============
          123
```

Function

If the parameter value entered in the function can be converted to the specified type, the parameter type will be converted. The strings are converted to numbers because the input parameter expected in the following function is a number.

```
SELECT MOD('123','2');

           mod('123', '2')
==========================
     1.000000000000000e+00
```

You can enter multiple type values in the function. If the type value not specified in the function is delivered, the type will be converted depending on the following priority order.

- Date/Time Type (**DATETIME** > **TIMESTAMP** > **DATE** > **TIME**)
- Approximate Numeric Type (**MONETARY** > **DOUBLE** > **FLOAT**)
- Exact Numeric Type (**NUMERIC** > **BIGINT** > **INT** > **SHORT**)
- String Type (**CHAR**/**NCHAR** > **VARCHAR**/**VARNCHAR**)

Comparison Operation

The following are the conversion rules according to an operand type of the comparison operator.

operand1 Type	operand2 Type	Conversion	Comparison
Numeric Type	Numeric Type	None	NUMERIC
	String Type	Converts operand2 to **DOUBLE**	NUMERIC
	Date/Time Type	None	N/a
String Type	Numeric Type	Converts operand1 to **DOUBLE**	NUMERIC
	String Type	None	String
	Date/Time Type	Converts operand1 to date/time type	Date/Time
Date/Time Type	Numeric Type	None	N/a
	String Type	Converts operand2 to date/time type	Date/Time
	Date/Time Type	Converts it to the type with higher priority	Date/Time

The following are the exceptions in the conversion rules for comparison operators:

- operand1 <operator> host variable

operand1 Type	operand2 Type	Conversion	Comparison
String Type	Numeric Type	Converts operand2 to the string type	String

- COLUMN <operator> value

operand1 Type	operand2 Type	Conversion	Comparison
String type	Numeric type	Converts operand2 to the string type	String
	Date/Time type	Converts operand2 to the string type	String

If operand2 is a set operator(**IS IN**, **IS NOT IN**, **= ALL**, **= ANY**, **< ALL**, **< ANY**, **<= ALL**, **<= ANY**, **>= ALL**, **>= ANY**), the exception above is not applied.

Numeric Type & String Type Operands

The string type operand will be converted to **DOUBLE**.

```
CREATE TABLE t(i INT, s STRING);
INSERT INTO t VALUES(1,'1'),(2,'2'),(3,'3'),(4,'4'), (12,'12');

SELECT i FROM t WHERE i < '11.3';
```

```
            i
=============
            1
            2
            3
            4

SELECT ('2' <= 11);

     ('2'<11)
=============
            1
```

String Type & Date/Time Type Operands

The string type operand will be converted to the date/time type.

```
SELECT ('2010-01-01' < date'2010-02-02');

   ('2010-01-01'<date '2010-02-02')
==================================
                                1

SELECT (date'2010-02-02' >= '2010-01-01');

  (date '2010-02-02'>='2010-01-01')
===================================
                                1
```

String Type & Numeric Type Host Variable Operands

The numeric type host variable will be converted to the string type.

```
PREPARE s FROM 'SELECT s FROM t WHERE s < ?';
EXECUTE s USING 11;
       s
===================
     '1'
```

String Type & Numeric Type value Operands

The numeric type value will be converted to the string type.

```
SELECT s FROM t WHERE s > 11;
       s
==================
     '2'
     '3'
     '4'
     '12'

SELECT s FROM t WHERE s BETWEEN 11 AND 33;
        s
======================
      '2'
      '3'
      '12'
```

String Type Column & Date/Time Type Value Operands

The date/time type value will be converted to the string type.

```
SELECT s FROM t;

           s
======================
    '01/01/1998'
    '01/01/1999'
    '01/01/2000'

SELECT s FROM t WHERE s <= date'02/02/1998';
```

```
            s
======================
    '01/01/1998'
    '01/01/1999'
    '01/01/2000'
```

Range Operation

Numeric Type and String Type Operands

The string type operand will be converted to **DOUBLE**.

```
SELECT i FROM t WHERE i <= all {'11','12'};

            i
=============
            1
            2
            3
            4
```

String Type and Date/Time Type Operands

The string type operand will be converted to the date/time type.

```
SELECT s FROM t2;

          s
======================
      '01/01/2000'
      '01/01/1999'
      '01/01/1998'

SELECT s FROM t2 WHERE s <= ALL {date'02/02/1998',date'01/01/2000'};

          s
======================
       '01/01/1998'
```

If it is impossible to convert to the corresponding type, an error is returned.

Arithmetic Operation

Date/Time Type Operand

If the date/time type operands are given to '-' operator and the types are different from each other, it will be converted to the type with a higher priority. The following example shows that the operand data type on the left is converted from **DATE** to **DATETIME**, so that the result of '-' operation of **DATETIME** can be displayed in milliseconds.

```
SELECT date'2002-01-01' - datetime'2001-02-02 12:00:00 am';

   date '2002-01-01'- datetime '2001-02-02 12:00:00 am'
=====================================================
                                          28771200000
```

Numeric Type Operand

If the numeric type operands are given and the types are different from each other, it will be converted to the type with the higher priority.

Date/Time Type & Numeric Type Operands

If the date/time type and the numeric type operands are given to '+' or '-' operator, the numeric type operand is converted to either **BIGINT**, **INT** or **SHORT**.

Date/Time Type & String Type Operands

If a date/time type and a string type are operands, only '+' and '-' operators are allowed. If the '+' operator is used, it will be applied according to the following rules.

- The string type will be converted to **BIGINT** with an interval value. The interval is the smallest unit for operands in the Date/Time type, and the interval for each type is as follows:
- **DATE** : Days
- **TIME**, **TIMESTAMP** : Seconds
- **DATETIME** : Milliseconds
- Floating-point numbers are rounded.
- The result type is the type of an date/time operand.

```
SELECT date'2002-01-01' + '10';

  date '2002-01-01'+'10'
======================
  01/11/2002
```

If the date/time type and a string type are operands and the '-' operator is used, they will be applied according to the following rules.

- If the date/time type operands are **DATE**, **DATETIME** and **TIMESTAMP**, the string will be converted to **DATETIME**; if the date/time operand is **TIME**, the string is converted to **TIME**.
- The result type is always **BIGINT**.

```
SELECT date'2002-01-01'-'2001-01-01';

  date '2002-01-01'-'2001-01-01'
================================
                    31536000000

-- this causes an error

SELECT date'2002-01-01'-'10';

 In line 1, column 13,
 ERROR: Cannot coerce '10' to type datetime.
```

Numeric Type & String Type Operands

If a numeric type and a string type are operands, they will be applied according to the following rules.

- Strings will be converted to **DOUBLE** when possible.
- The result type is **DOUBLE** or **MONETARY** and depends on the type of the numeric operand.

```
SELECT 4 + '5.2';

                4+'5.2'
==========================
  9.199999999999999e+00
```

Unlike CUBRID 2008 R3.1 and the earlier versions, the string in the date/time format, that is, the string such as '2010-09-15' is not converted to the date/time type. You can use a literal (DATE'2010-09-15') with the date/time type for addition and subtraction operations.

```
SELECT '2002-01-01'+1;
   ERROR: Cannot coerce '2002-01-01' to type double.
SELECT DATE'2002-01-01'+1;
  date '2002-01-01'+1
=====================
  01/02/2002
```

String Type Operand

If you multiply, divide or subtract both strings, the result returns a **DOUBLE** type value.

```
SELECT '3'*'2';
```

```
                         '3'*'2'
=============================
       6.000000000000000e+00
```

The '+' operator action depends on how to set the system parameter **plus_as_concat** in the **cubrid.conf** file. For more information, see Syntax/Type Related Parameter.

- If a value for **plus_as_concat** is yes, the concatenation of two strings will be returned.

```
SELECT '1'+'1';

                 '1'+'1'
======================
                    '11'
```

- If a value for **plus_as_concat** is no and two strings can be converted to numbers, the **DOUBLE** type value will be returned by adding the two numbers.

```
SELECT '1'+'1';

                     '1'+'1'
==========================
     2.000000000000000e+00
```

If it is impossible to convert to the corresponding type, an error is returned.

Table Definition

CREATE TABLE

Table Definition

Description

To create a table, use the **CREATE TABLE** syntax.

Syntax

```
CREATE {TABLE | CLASS} <table_name>
                   [ <subclass_definition> ]
                   [ ( <column_definition> [,<table_constraint>]... ) ]
                   [ AUTO_INCREMENT = initial_value ] ]
                   [ CLASS ATTRIBUTE ( <column_definition_comma_list> ) ]
                   [ METHOD <method_definition_comma_list> ]
                   [ FILE <method_file_comma_list> ]
                   [ INHERIT <resolution_comma_list> ]
                   [ REUSE_OID ]
<column_definition> ::=
column_name column_type [[ <default_or_shared> ] | [ <column_constraint> ]]...

<default_or_shared> ::=
{SHARED <value_specification> | DEFAULT <value_specification> } |
AUTO_INCREMENT [(seed, increment)]

<column_constraint> ::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY <referential definition>

<table_constraint> ::=
[ CONSTRAINT [ <constraint_name> ] ] UNIQUE [ KEY | INDEX ]( column_name_comma_list ) |
[ { KEY | INDEX } [ <constraint_name> ]( column_name_comma_list ) |
[ PRIMARY KEY ( column_name_comma_list )] |
[ <referential_constraint> ]

<referential_constraint> ::=
FOREIGN KEY [ <foreign_key_name> ]( column_name_comma_list ) <referential definition>

<referential definition> ::=
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ <referential_triggered_action> ... ]

<referential_triggered_action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL

<subclass_definition> ::=
{ UNDER | AS SUBCLASS OF } table_name_comma_list

<method_definition> ::=
[ CLASS ] method_name
[ ( [ argument_type_comma_list ] ) ]
[ result_type ]
[ FUNCTION  function_name ]

<resolution> ::=
[ CLASS ] { column_name | method_name } OF superclass_name
[ AS alias ]
```

- *table_name* : Specifies the name of the table to be created (maximum : 255 bytes).
- *column_name* : Specifies the name of the column to be created.

- *column_type* : Specifies the data type of the column.
- [**SHARED** *value* | **DEFAULT** *value*] : Specifies the initial value of the column (see Column Definition for more information).
- *column_constraints* : Specifies the constraint of the column. Available constraints are **NOT NULL**, **UNIQUE**, **PRIMARY KEY** and **FOREIGN KEY** (see Constraint Definition for more information).

Example

```
CREATE TABLE olympic (
    host_year               INT       NOT NULL PRIMARY KEY,
    host_nation           VARCHAR(40)  NOT NULL,
    host_city               VARCHAR(20)  NOT NULL,
    opening_date         DATE                NOT NULL,
    closing_date         DATE                NOT NULL,
    mascot                      VARCHAR(20) ,
    slogan                      VARCHAR(40) ,
    introduction         VARCHAR(1500)
)
```

Column Definition

A column is a set of data values of a particular simple type, one for each row of the table.

```
<column_definition> ::=
column_name column_type [ [ <default_or_shared> ] | [ <column_constraint> ] ]...

<default_or_shared> ::=
{ SHARED <value_specification> | DEFAULT <value_specification> } |
AUTO_INCREMENT [ (seed, increment) ]

<column_constraint> ::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY <referential definition>
```

Column Name

Description

How to create a column name, see Identifier.

You can alter created column name by using RENAME COLUMN clause of the **ALTER TABLE**.

Example

The following is an example of creating the manager2 table that has the following two columns: full_name and age.

```
CREATE TABLE manager2 (full_name VARCHAR(40), age INT );
```

Caution

- The first character of a column name must be an alphabet. The maximum length is 255 characters.
- The column name must be unique in the table.

Setting the Column Initial Value (SHARED, DEFAULT)

Description

SHARED and **DEFAULT** are attributes related to the initial value of the column. You can change the value of **SHARED** and **DEFAULT** in the **ALTER TABLE** statement.

- **SHARED** : Column values are identical in all rows. If a value different from the initial value is **INSERT**ed, the column value is updated to a new one in every row.
- **DEFAULT** : The initial value set when the **DEFAULT** attribute was defined is saved even if the column value is not specified when a new row is inserted. Note that if you set **SYS_TIMESTAMP** as a **DEFAULT** value when creating a table, the **TIMESTAMP** value at the point of **CREATE TABLE**, not the point at which the data is **INSERT**ed, is specified by default. Therefore, you must specify the **SYS_TIMESTAMP** value for the **VALUES** of the **INSERT** statement when entering data.

Example

```
CREATE TABLE colval_tbl
( id INT, name VARCHAR SHARED 'AAA', phone VARCHAR DEFAULT '000-0000');
INSERT INTO colval_tbl(id) VALUES (1),(2);
SELECT * FROM colval_tbl;

           id  name                  phone
=========================================================
            1  'AAA'                 '000-0000'
            2  'AAA'                 '000-0000'

--updating column values on every row
INSERT INTO colval_tbl(id, name) VALUES (3,'BBB');
INSERT INTO colval_tbl(id) VALUES (4),(5);
SELECT * FROM colval_tbl;

           id  name                  phone
=========================================================
            1  'BBB'                 '000-0000'
            2  'BBB'                 '000-0000'
            3  'BBB'                 '000-0000'
            4  'BBB'                 '000-0000'
            5  'BBB'                 '000-0000'

--changing DEFAULT value in the ALTER TABLE statement
ALTER TABLE colval_tbl CHANGE phone DEFAULT '111-1111'
INSERT INTO colval_tbl(id) VALUES (6);
SELECT * FROM colval_tbl;

           id  name                  phone
=========================================================
            1  'BBB'                 '000-0000'
            2  'BBB'                 '000-0000'
            3  'BBB'                 '000-0000'
            4  'BBB'                 '000-0000'
            5  'BBB'                 '000-0000'
            6  'BBB'                 '111-1111'
```

AUTO INCREMENT

Description

You can define the **AUTO_INCREMENT** attribute for the column to automatically give serial numbers to column values. This can be defined only for **SMALLINT**, **INTEGER**, **BIGINT**(*p*,0), and **NUMERIC**(*p*,0) domains.

DEFAULT, **SHARED** and **AUTO_INCREMENT** cannot be defined for the same column. Make sure the value entered directly by the user and the value entered by the auto increment attribute do not conflict with each other.

You can change the initial value of **AUTO_INCREMENT** by using the **ALTER TABLE** statement. For more information, see AUTO_INCREMENT Statement of **ALTER TABLE**.

Syntax

```
CREATE TABLE table_name (id int AUTO_INCREMENT[(seed, increment)]) |
CREATE TABLE table_name (id int AUTO_INCREMENT) AUTO_INCREMENT = seed;
```

- *seed* : The initial value from which the number starts. Only positive integers are allowed. The default is 1.
- *increment* : The increment value of each row. Only positive integers are allowed. The default value 1.

When you use the **CREATE TABLE** *table_name* (id int **AUTO_INCREMENT**) **AUTO_INCREMENT** = *seed*; statement, the constraints are as follows:

- You should define only one column with the **AUTO_INCREMENT** attribute.
- Don't use (*seed*, increment) and `AUTO_INCREMENT` = *seed* together.

Example

```
CREATE TABLE auto_tbl(id INT AUTO_INCREMENT, name VARCHAR);
INSERT INTO auto_tbl VALUES(NULL, 'AAA'),(NULL, 'BBB'),(NULL, 'CCC');
```

```
INSERT INTO auto_tbl(name) VALUES ('DDD'),('EEE');
SELECT * FROM auto_tbl;

           id  name
===================================
            1  'AAA'
            2  'BBB'
            3  'CCC'
            4  'DDD'
            5  'EEE'

CREATE TABLE tbl (id int AUTO_INCREMENT, val string) AUTO_INCREMENT = 3;
INSERT INTO tbl VALUES (NULL,'cubrid');

SELECT * FROM tbl;
           id  val
===================================
            3  'cubrid'

CREATE TABLE t (id int AUTO_INCREMENT, id2 int AUTO_INCREMENT) AUTO_INCREMENT = 5; ERROR:
To avoid ambiguity, the AUTO_INCREMENT table option requires the table to  have exactly
one AUTO_INCREMENT column and no seed/increment specification.
CREATE TABLE t (i int AUTO_INCREMENT(100, 2)) AUTO_INCREMENT = 3; ERROR: To avoid
ambiguity, the AUTO_INCREMENT table option requires the table to  have exactly one
AUTO_INCREMENT column and no seed/increment specification.
```

Caution

- Even if a column has auto increment, the **UNIQUE** constraint is not satisfied.
- If **NULL** is specified in the column where auto increment is defined, the value of auto increment is stored.
- The initial value and the final value obtained by auto increment cannot exceed the minimum and maximum values allowed in the given domain.
- Because auto increment has no cycle, an error occurs when the maximum value of the type exceeds, and no rollback is executed. Therefore, you must delete and recreate the column in such cases.
- For example, if a table is created as below, the maximum value of A is 32767. Because an error occurs if the value exceeds 32767, you must make sure that the maximum value of the column A does not exceed the maximum value of the type when creating the initial table.

```
create table tbl(A smallint auto_increment, B char(5));
```

Constraint Definition

Description

You can define **NOT NULL**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY** as the constraints. You can also create an index by using **INDEX** or **KEY**.

```
<column_constraint> ::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY <referential definition>

<table_constraint> ::=
[ CONSTRAINT [ <constraint_name> ] ] UNIQUE [ KEY | INDEX ]( column_name_comma_list ) |
[ { KEY | INDEX } [ <constraint_name> ]( column_name_comma_list ) |
[ PRIMARY KEY ( column_name_comma_list )] |
[ <referential_constraint> ]

<referential_constraint> ::=
FOREIGN KEY ( column_name_comma_list ) <referential definition>

<referential definition> ::=
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ <referential_triggered_action> ... ]

<referential_triggered_action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
```

```
CASCADE | RESTRICT | NO ACTION  | SET NULL
```

NOT NULL Constraints

Description

A column for which the **NOT NULL** constraint has been defined must have a certain value that is not **NULL**. The **NOT NULL** constraint can be defined for all columns. An error occurs if you try to insert a **NULL** value into a column with the **NOT NULL** constraint by using the **INSERT** or **UPDATE** statement.

Example

```
CREATE TABLE const_tbl1(id INT NOT NULL, INDEX i_index(id ASC), phone VARCHAR);

CREATE TABLE const_tbl2(id INT NOT NULL PRIMARY KEY, phone VARCHAR);
INSERT INTO const_tbl2 (NULL,'000-0000');

In line 2, column 25,

ERROR: syntax error, unexpected Null
```

UNIQUE Constraint

Description

The **UNIQUE** constraint enforces a column to have a unique value. An error occurs if a new record that has the same value as the existing one is added by this constraint.

You can place a **UNIQUE** constraint on either a column or a set of columns. If the **UNIQUE** constraint is defined for multiple columns, the uniqueness is ensured not for each column, but the combination of multiple columns.

Example

If a **UNIQUE** constraint is defined on a set of columns, this ensures the uniqueness of the values in all the columns. As shown below, the second INSERT statement succeeds because the value of column a is the same, but the value of column b is unique. The third INSERT statement causes an error because the values of column a and b are the same as those in the first INSERT statement.

```
--UNIQUE constraint is defined on a single column only
CREATE TABLE const_tbl5(id INT UNIQUE, phone VARCHAR);
INSERT INTO const_tbl5(id) VALUES (NULL), (NULL);
INSERT INTO const_tbl5 VALUES (1, '000-0000');
SELECT * FROM const_tbl5;

           id  phone
===================================
         NULL  NULL
         NULL  NULL
            1  '000-0000'

INSERT INTO const_tbl5 VALUES (1, '111-1111');

ERROR: Operation would have caused one or more unique constraint violations.


--UNIQUE constraint is defined on several columns
CREATE TABLE const_tbl6(id INT, phone VARCHAR, CONSTRAINT UNIQUE(id,phone));
INSERT INTO const_tbl6 VALUES (1,NULL), (2,NULL), (1,'000-0000'), (1,'111-1111');
SELECT * FROM const_tbl6;

           id  phone
===================================
            1  NULL
            2  NULL
            1  '000-0000'
            1  '111-1111'
```

PRIMARY KEY Constraint

Description

A key in a table is a set of column(s) that uniquely identifies each row. A candidate key is a set of columns that uniquely identifies each row of the table. You can define one of such candidate keys a primary key. That is, the column defined as a primary key is uniquely identified in each row.

By default, the index created by defining the primary key is created in ascending order, and you can define the order by specifying **ASC** or **DESC** keyword next to the column.

Syntax

```
CREATE TABLE pk_tbl (a INT, b INT, PRIMARY KEY (a, b DESC));
```

Example

```
CREATE TABLE const_tbl7(
id INT NOT NULL,
phone VARCHAR,
CONSTRAINT pk_id PRIMARY KEY(id));

--CONSTRAINT keyword
CREATE TABLE const_tbl8(
id INT NOT NULL PRIMARY KEY,
phone VARCHAR);

--primary key is defined on multiple columns
CREATE TABLE const_tbl8 (
host_year    INT NOT NULL,
event_code   INT NOT NULL,
athlete_code INT NOT NULL,
medal        CHAR(1)  NOT NULL,
score        VARCHAR(20),
unit         VARCHAR(5),
PRIMARY KEY(host_year, event_code, athlete_code, medal)
);
```

FOREIGN KEY Constraint

Description

A foreign key is a column or a set of columns that references the primary key in other tables in order to maintain reference relationship. The foreign key and the referenced primary key must have the same data type. Consistency between two tables is maintained by the foreign key referencing the primary key, which is called referential integrity.

Syntax

```
[ CONSTRAINT < constraint_name > ]
FOREIGN KEY [ <foreign_key_name> ] ( column_name_comma_list )
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ <referential_triggered_action> ]

<referential_triggered_action> :
ON UPDATE <referential_action>
[ ON DELETE <referential_action> [ ON CACHE OBJECT cache_object_column_name ]]

<referential_action> :
CASCADE | RESTRICT | NO ACTION | SET NULL
```

- *constraint_name* : Specifies the name of the table to be created.
- *foreign_key_name* : Specifies a name of the **FOREIGN KEY** constraint. You can skip the name specification. However, if you specify this value, *constraint_name* will be ignored, and the specified value will be used.
- *column_name* : Specifies the name of the column to be defined as a foreign key after the **FOREIGN KEY** keyword. There is no limit on the number of foreign keys to be defined (the number of columns), but it must be the same number as that of the referred primary keys.
- *referenced_table_name* : Specifies the name of the table to be referenced.

- *column_name* : Specifies the name of the referred primary key column after the **FOREIGN KEY** keyword.
- *referential_triggered_action* : Specifies the trigger action that responds to a certain operation in order to maintain referential integrity. **ON UPDATE**, **ON DELETE** or **ON CACHE OBJECT** can be specified. Each action can be defined multiple times, and the definition order is not significant.
- **ON UPDATE** : Defines the action to be performed when attempting to update the primary key referenced by the foreign key. You can use either **NO ACTION**, **RESTRICT**, or **SET NULL** option. The default is **RESTRICT**.
- **ON DELETE** : Defines the action to be performed when attempting to delete the primary key referenced by the foreign key. You can use **NO ACTION**, **RESTRICT**, **CASCADE**, or **SET NULL** option. The default is **RESTRICT**.
- **ON CACHE OBJECT** : You can search an object using a direct object reference in object-oriented model. **ON CACHE OBJECT** option supports this feature in association with referential integrity (foreign key). **ON CACHE OBJECT** option adds an OID reference to a foreign key configuration. The OID is used as a CACHE point for the foreign key to the primary key table. Such OID is managed by the system internally; it cannot be changed by users. To define the **ON CACHE OBJECT** option, you must have defined a column whose domain is the table with a primary key and specified the column in the *cache_object_column_name*.
  The attribute defined with **ON CACHE OBJECT** can use the OID the same way as the one of the existing object type.
- *referential_ action* : You can define an option that determines whether to maintain the value of the foreign key when the primary key value is deleted or updated.
- **CASCADE** : If the primary key is deleted, the foreign key is deleted as well. This option is supported only for the **ON DELETE** operation.
- **RESTRICT** : Prevents the value of the primary key from being deleted or updated, and rolls back any transaction that has been attempted.
- **SET NULL** : When a specific record is being deleted or updated, the column value of the foreign key is updated to **NULL**.
- **NO ACTION** : Its behavior is the same as that of the **RESTRICT** option.

Example

```
--creaing two tables where one is referencing the other
CREATE TABLE a_tbl(
id INT NOT NULL DEFAULT 0 PRIMARY KEY,
phone VARCHAR(10));

CREATE TABLE b_tbl(
ID INT NOT NULL,
name VARCHAR(10) NOT NULL,
CONSTRAINT pk_id PRIMARY KEY(id),
CONSTRAINT fk_id FOREIGN KEY(id) REFERENCES a_tbl(id)
ON DELETE CASCADE ON UPDATE RESTRICT);

INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333');
INSERT INTO b_tbl VALUES(1,'George'),(2,'Laura'),(3,'Max');
SELECT a.id, b.id, a.phone, b.name FROM a_tbl a, b_tbl b WHERE a.id=b.id;

           id           id                    phone                 name
==============================================================================
            1            1                    '111-1111'            'George'
            2            2                    '222-2222'            'Laura'
            3            3                    '333-3333'            'Max'

--when deleting primay key value, it cascades foreign key value
DELETE FROM a_tbl WHERE id=3;

1 rows affected.

SELECT a.id, b.id, a.phone, b.name FROM a_tbl a, b_tbl b WHERE a.id=b.id;

           id           id                    phone                 name
==============================================================================
            1            1                    '111-1111'            'George'
            2            2                    '222-2222'            'Laura'

--when attempting to update primay key value, it restricts the operation
UPDATE  a_tbl SET id = 10 WHERE phone = '111-1111';
```

```
In the command from line 1,

ERROR: Update/Delete operations are restricted by the foreign key 'fk_id'.

0 command(s) successfully processed.
```

Caution

- In a referential constraint, the name of the primary key table to be referenced and the corresponding column names are defined. If the list of column names are is not specified, the primary key of the primary key table is specified in the defined order.
- The number of primary keys in a referential constraint must be identical to that of foreign keys. The same column name cannot be used multiple times for the primary key in the referential constraint.
- The actions cascaded by reference constraints do not activate the trigger action.
- It is not recommended to use **referential_triggered_action** in the CUBRID HA environment. In the CUBRID HA environment, the trigger action is not supported. Therefore, if you use **referential_triggered_action**, the data between the master database and the slave database can be inconsistent. For more information, see CUBRID HA.

KEY or INDEX

Description

KEY and **INDEX** are used interchangeably. They create an index that uses the corresponding column as a key. You can specify the index name. If omitted, a name is assigned automatically.

Example

```
CREATE TABLE const_tbl3(id INT, phone VARCHAR, INDEX(id DESC, phone ASC));

CREATE TABLE const_tbl4(id INT, phone VARCHAR, KEY i_key(id DESC, phone ASC));
```

Column Option

Description

You can specify options such as **ASC** or **DESC** after the column name when defining **UNIQUE** or **INDEX** for a specific column. This keyword is specified to save the index value in ascending or descending order.

Syntax

```
column_name [ASC|DESC]
```

Example

```
CREATE TABLE const_tbl(
id VARCHAR,
name VARCHAR,
CONSTRAINT UNIQUE INDEX(id DESC, name ASC)
);

INSERT INTO const_tbl VALUES('1000', 'john'), ('1000','johnny'), ('1000', 'jone');
INSERT INTO const_tbl VALUES('1001', 'johnny'), ('1001','john'), ('1001', 'jone');

SELECT * FROM const_tbl WHERE id > '100';
===================================================
          id    name
          1001     john
          1001     johnny
          1001     jone
          1000     john
          1000     johnny
          1000     jone
```

Table Option (REUSE_OID)

Description

You can specify the **REUSE_OID** option when creating a table, so that OIDs that have been deleted due to the deletion of records (**DELETE**) can be reused when a new record is inserted (**INSERT**). Such a table is called an OID reusable or a non-referable table.

OID (Object Identifier) is an object identifier represented by physical location information such as the volume number, page number and slot number. By using such OIDs, CUBRID manages the reference relationships of objects and searches, saves or deletes them. When an OID is used, accessibility is improved because the object in the heap file can be directly accessed without referring to the table. However, the problem of decreased reusability of the storage occurs when there are many **DELETE/ INSERT** operations because the object's OID is kept to maintain the reference relationship with the object even if it is deleted.

If you specify the **REUSE_OID** option when creating a table, the OID is also deleted when data in the table is deleted, so that another **INSERT**ed data can use it. OID reusable tables cannot be referred to by other tables, and OID values of the objects in the OID reusable tables cannot be viewed.

Example

```
--creating table with REUSE_OID option specified
CREATE TABLE reuse_tbl (a INT PRIMARY KEY) REUSE_OID;
INSERT INTO reuse_tbl VALUES (1);
INSERT INTO reuse_tbl VALUES (2);
INSERT INTO reuse_tbl VALUES (3);

--an error occurs when column type is a OID reusable table itself
CREATE TABLE tbl_1 ( a reuse_tbl);

ERROR: The class 'reuse_tbl' is marked as REUSE_OID and is non-referable. Non-referable
classes can't be the domain of an attribute and their instances' OIDs cannot be returned.

--an error occurs when a table references a OID reusable table
CREATE TABLE tbl_2
(b int, FOREIGN KEY(b) REFERENCES reuse_tbl(a) ON CACHE OBJECT oid_value);
INSERT INTO tbl_2(b) VALUES(1);
SELECT oid_value.a FROM tbl_2;

ERROR: The class 'reuse_tbl' is marked as REUSE_OID and is non-referable. Non-referable
classes can't be the domain of an attribute and their instances' OIDs cannot be returned.
```

Caution

- OID reusable tables cannot be referred to by other tables.
- Updatable views cannot be created for OID reusable tables.
- OID reusable tables cannot be specified as class attribute domains of other tables.
- OID values of the objects in the OID reusable tables cannot be read.
- Instance methods cannot be called from OID reusable tables. Also, instance methods cannot be called if a subclass inherited from the class where the method is defined is defined as an OID reusable table.
- OID reusable tables are supported only by CUBRID 2008 R2.2 or above, and backward compatibility is not ensured. That is, the database in which the OID reusable table is located cannot be accessed from a lower version database.
- OID reusable tables can be managed as partitioned tables and can be replicated.

CREATE TABLE LIKE

Description

You can create a table that has the same schema as an existing table by using the **CREATE TABLE...LIKE** statement.

Column attribute, table constraint, and index are replicated from the existing table. An index name created from the existing table changes according to a new table name, but an index name defined by a user is replicated as it is.

Therefore, you should be careful at a query statement that is supposed to use a specific index created by using the **USING INDEX**.

You cannot create the column definition because the **CREATE TABLE...LIKE** statement replicates the schema only.

Syntax

```
CREATE {TABLE | CLASS} <new_table_name> LIKE <old_table_name>
```

- *new_table_name* : A table name to be created.
- *old_table_name* : The name of the original table that already exists in the database. The following tables cannot be specified as original tables in the **CREATE TABLE…LIKE** statement.
- Partition table
- Table that contains an **AUTO_INCREMENT** column
- Table that uses inheritance or methods

Example

```
CREATE TABLE a_tbl(
id INT NOT NULL DEFAULT 0 PRIMARY KEY,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333');

--creating an empty table with the same schema as a_tbl
CREATE TABLE new_tbl LIKE a_tbl;
SELECT * FROM new_tbl;

There are no results.

;schema a_tbl

=== <Help: Schema of a Class> ===


 <Class Name>

     a_tbl

 <Attributes>

     id                   INTEGER DEFAULT 0 NOT NULL
     phone                CHARACTER VARYING(10)

 <Constraints>

     PRIMARY KEY pk_a_tbl_id ON a_tbl (id)

Current transaction has been committed.

;schema new_tbl

=== <Help: Schema of a Class> ===


 <Class Name>

     new_tbl

 <Attributes>

     id                   INTEGER DEFAULT 0 NOT NULL
     phone                CHARACTER VARYING(10)

 <Constraints>

     PRIMARY KEY pk_new_tbl_id ON new_tbl (id)


Current transaction has been committed.
```

CREATE TABLE AS SELECT

Description

You can create a new table that contains the result records of the **SELECT** statement by using the **CREATE TABLE...AS SELECT** statement. You can define column and table constraints for the new table. The following rules are applied to reflect the result records of the **SELECT** statement.

- If *col_1* is defined in the new table and the same column *col_1* is specified in select_statement, the result record of the **SELECT** statement is stored as *col_1* value in the new table. Type casting is attempted if the column names are identical but the columns types are different.
- If *col_1* and *col_2* are defined in the new table, *col_1*, col_2 and *col_3* are specified in the column list of the *select_statement* and there is a containment relationship between all of them, *col_1*, *col_2* and *col_3* are created in the new table and the result data of the **SELECT** statement is stored as values for all columns. Type casting is attempted if the column names are identical but the columns types are different.
- If columns *col_1* and *col_2* are defined in the new table and *col_1* and *col_3* are defined in the column list of *select_statement* without any containment relationship between them, *col_1*, *col_2* and *col_3* are created in the new table, the result data of the **SELECT** statement is stored only for *col_1* and *col_3* which are specified in *select_statement*, and **NULL** is stored as the value of *col_2*.
- Column aliases can be included in the column list of *select_statement*. In this case, new column alias is used as a new table column name. It is recommended to use an alias because invalid column name is created, if an alias does not exist when a function calling or an expression is used.
- The **REPLACE** option is valid only when the **UNIQUE** constraint is defined in a new table column (*col_1*). When duplicate values exist in the result record of *select_statement*, a **UNIQUE** value is stored for *col_1* if the **REPLACE** option has been defined, or an error message is displayed if the **REPLACE** option is omitted due to the violation of the **UNIQUE** constraint.

Syntax

```
CREATE {TABLE | CLASS} <table_name>
                   [( <column_definition> [,<table_constraint>]... )]
                   [REPLACE] AS <select_statement>
```

- *table_name* : A name of the table to be created.
- *column_definition* : Defines a column. If it is omitted, the column schema of **SELECT** statement is replicated; however, the constraint or the **AUTO_INCREMENT** attribute is not replicated.
- *table_constraint* : Defines table constraint.
- *select_statement* : A **SELECT** statement targeting a source table that already exists in the database.

Example

```
CREATE TABLE a_tbl(
id INT NOT NULL DEFAULT 0 PRIMARY KEY,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333');

--creating a table without column definition
CREATE TABLE new_tbl1 AS SELECT * FROM a_tbl;
SELECT * FROM new_tbl1;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'

--all of column values are replicated from a_tbl
CREATE TABLE new_tbl2
(id INT NOT NULL AUTO_INCREMENT PRIMARY KEY, phone VARCHAR) AS SELECT * FROM a_tbl;
SELECT * FROM new_tbl2;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
```

```
--some of column values are replicated from a_tbl and the rest is NULL
CREATE TABLE new_tbl3
(id INT, name VARCHAR) AS SELECT id, phone FROM a_tbl;
SELECT * FROM new_tbl3

  name                           id  phone
=========================================================
  NULL                            1  '111-1111'
  NULL                            2  '222-2222'
  NULL                            3  '333-3333'

--column alias in the select statement should be used in the column definition
CREATE TABLE new_tbl4
(id1 int, id2 int)AS SELECT t1.id id1, t2.id id2 FROM new_tbl1 t1, new_tbl2 t2;
SELECT * FROM new_tbl4;

          id1          id2
==========================
            1            1
            1            2
            1            3
            2            1
            2            2
            2            3
            3            1
            3            2
            3            3

--REPLACE is used on the UNIQUE column
CREATE TABLE new_tbl5(id1 int UNIQUE) REPLACE AS SELECT * FROM new_tbl4;
SELECT * FROM new_tbl5;

          id1          id2
==========================
            1            3
            2            3
            3            3
```

ALTER TABLE

Overview

Description

You can modify the structure of a table by using the **ALTER** statement. You can perform operations on the target table such as adding/deleting columns, creating/deleting indexes, and type casting existing columns as well as changing table names, column names and constraints. **TABLE** and **CLASS** are used interchangeably **VIEW** and **VCLASS**, and **COLUMN** and **ATTRIBUTE** as well.

You can also change the initial value of **AUTO_INCREMENT**.

Syntax

```
ALTER [ <class_type> ] <table_name> <alter_clause> ;

<class_type> ::= TABLE | CLASS | VCLASS | VIEW

<alter_clause> ::= ADD <alter_add> [ INHERIT <resolution_comma_list> ] |
                   ADD { KEY | INDEX } [index_name] (<index_col_name>) |
                   ALTER [ COLUMN ] column_name SET DEFAULT <value_specifiation> |
                   DROP <alter_drop> [ INHERIT <resolution_comma_list> ] |
                   DROP { KEY | INDEX } index_name |
                   DROP FOREIGN KEY constraint_name |
                   DROP PRIMARY KEY |
                   RENAME <alter_rename> [ INHERIT <resolution_comma_list> ] |
                   CHANGE <alter_change> |
                   INHERIT <resolution_comma_list>
                   AUTO_INCREMENT = <initial_value>
```

```
<alter_add> ::= [ ATTRIBUTE | COLUMN ] [(]<class_element_comma_list>[)] [ FIRST | AFTER
old_column_name ] |
                CLASS ATTRIBUTE <column_definition_comma_list> |
                CONSTRAINT < constraint_name > <column_constraint> ( column_name )|
                FILE <file_name_comma_list> |
                METHOD <method_definition_comma_list> |
                QUERY <select_statement> |
                SUPERCLASS <class_name_comma_list>

<alter_change> ::= FILE <file_path_name> AS <file_path_name> |
                   METHOD <method_definition_comma_list> |
                   QUERY [ <unsigned_integer_literal> ] <select_statement> |
                   <column_name> DEFAULT <value_specification>

<alter_drop> ::= [ ATTRIBUTE | COLUMN | METHOD ]
                 <column_name_comma_list> |
                 FILE <file_name_comma_list> |
                 QUERY [ <unsigned_integer_literal> ] |
                 SUPERCLASS <class_name_comma_list> |
                 CONSTRAINT <constraint_name>

<alter_rename> ::= [ ATTRIBUTE | COLUMN | METHOD ]
                   <old_column_name> AS <new_column_name> |
                   FUNCTION OF <column_name> AS <function_name>
                   FILE <file_path_name> AS <file_path_name>

<resolution> ::= { column_name | method_name } OF <superclass_name>
                 [ AS alias ]

<class_element> ::= <column_definition> | <table_constraint>

<column_constraint> ::= UNIQUE [ KEY ] | PRIMARY KEY | FOREIGN KEY

<index_col_name> ::=
column_name [(length)] [ ASC | DESC ]
```

Caution

The table name can be changed only by the table owner, **DBA** and **DBA** members. The other users must be granted to change the name by the owner or **DBA** (see Granting Authorization for more information on authorization).

ADD COLUMN Clause

Description

You can add a new column by using the **ADD COLUMN** clause. You can specify the location of the column to be added by using the **FIRST** or **AFTER** keyword.

If the newly added column has the **NOT NULL** constraint but no **DEFAULT** constraint, it will have the hard default when the database server configuration parameter, **add_column_update_hard_default** is set to yes. However, when the parameter is set to no, the column will have **NULL** even with the **NOT NULL** constraint.

If the newly added column has the **PRIMARY KEY** or **UNIQUE** constraints, an error will be returned when the database server configuration parameter **add_column_update_hard_default** is set to yes. When the parameter is set to no, all data will have **NULL**. The default value of **add_column_update_hard_default** is **no**.

For **add_column_update_hard_default** and the hard default, see CHANGE Clause.

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
ADD [ COLUMN | ATTRIBUTE ] [(]<column_definition>[)] [ FIRST | AFTER old_column_name ]

column_definition ::=
column_name column_type
    { [ NOT NULL | NULL ] |
      [ { SHARED <value_specification> | DEFAULT <value_specification> }
          | AUTO_INCREMENT [(seed, increment)] ] |
```

```
      [ UNIQUE [ KEY ] |
          [ PRIMARY KEY | FOREIGN KEY REFERENCES
              [ referenced_table_name ]( column_name_comma_list )
              [ <referential_triggered_action> ... ]
          ]
      ] } ...

<referential_triggered_action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL
```

- *table_name* : Specifies the name of a table that has a column to be added.
- *column_definition* : Specifies the name, data type, and constraints of a column to be added.
- **AFTER** *oid_column_name* : Specifies the name of an existing column before the column to be added.

Example

```
CREATE TABLE a_tbl;
ALTER TABLE a_tbl ADD COLUMN age INT DEFAULT 0 NOT NULL;
INSERT INTO a_tbl(age) VALUES(20),(30),(40);
ALTER TABLE a_tbl ADD COLUMN name VARCHAR FIRST;
ALTER TABLE a_tbl ADD COLUMN id INT NOT NULL AUTO_INCREMENT UNIQUE;
ALTER TABLE a_tbl ADD COLUMN phone VARCHAR(13) DEFAULT '000-0000-0000' AFTER name;

SELECT * FROM a_tbl;

  name                  phone                         age           id
======================================================================
  NULL                  '000-0000-0000'                20         NULL
  NULL                  '000-0000-0000'                30         NULL
  NULL                  '000-0000-0000'                40         NULL

--adding multiple columns
ALTER TABLE a_tbl ADD COLUMN (age1 int, age2 int, age3 int);
```

ADD CONSTRAINT Clause

Description

You can add a new constraint by using the **ADD CONSTRAINT** clause.

By default, the index created when you add **PRIMARY KEY** constraints is created in ascending order, and you can define the key sorting order by specifying the **ASC** or **DESC** keyword next to the column name.

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
ADD CONSTRAINT < constraint_name > column_constraint ( column_name_comma_list )

column_constraint ::=
UNIQUE [ KEY ] |
PRIMARY KEY |
FOREIGN KEY [ <foreign_key_name> ] REFERENCES
[referenced_table_name]( column_name_comma_list )
                       [ <referential_triggered_action> ... ]

<referential_triggered_action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL
```

- *table_name* : Specifies the name of a table that has a constraint to be added.

- *constraint_name* : Specifies the name of a constraint to be added, or it can be omitted. If omitted, a name is automatically assigned.
- *foreign_key_name* : Specifies a name of the **FOREIGN KEY** constraint. You can skip the name specification. However, if you specify this value, *constraint_name* will be ignored, and the specified value will be used.
- *column_constraint* : Defines a constraint for the specified column. For more information, see Constraint Definition.

Example

```
ALTER TABLE a_tbl ADD CONSTRAINT PRIMARY KEY(id);
ALTER TABLE a_tbl ADD CONSTRAINT PRIMARY KEY(id, no DESC);
ALTER TABLE a_tbl ADD CONSTRAINT UNIQUE u_key1(id);
```

ADD INDEX Clause

Description

You can define the index attributes for a specific column by using the **ADD INDEX** clause.

Syntax

```
ALTER [ TABLE | CLASS ] table_name ADD { KEY | INDEX } [index_name] (<index_col_name>)

<index_col_name> ::=
column_name [(length)] [ ASC | DESC ]
```

- *table_name* : Specifies the name of a table to be modified.
- *index_name* : Specifies the name of an index. If omitted, a name is automatically assigned.
- *index_col_name* : Specifies the column that has an index to be defined. **ASC** or **DESC** can be specified for a column option; *prefix_length* of an index key also can be specified for a column option.

Example

```
ALTER TABLE a_tbl ADD INDEX (age ASC), ADD INDEX(phone DESC);
;schema a_tbl

=== <Help: Schema of a Class> ===

 <Class Name>

     a_tbl

<Attributes>

     name                    CHARACTER VARYING(1073741823) DEFAULT ''
     phone                   CHARACTER VARYING(13) DEFAULT '111-1111'
     age                     INTEGER
     id                      INTEGER AUTO_INCREMENT  NOT NULL

 <Constraints>

     UNIQUE u_a_tbl_id ON a_tbl (id)
     INDEX i_a_tbl_age ON a_tbl (age)
     INDEX i_a_tbl_phone_d ON a_tbl (phone DESC)

Current transaction has been committed.
```

ALTER COLUMN … SET DEFAULT Clause

Description

You can specify a new default value for a column that has no default value or modify the existing default value by using the **ALTER COLUMN** … **SET DEFAULT**. You can use the **CHANGE** clause to change the default value of multiple columns with a single statement. For more information, see the CHANGE Clause.

Syntax

```
ALTER [ TABLE | CLASS ] table_name ALTER [COLUMN] column_name SET DEFAULT value
```

- *table_name* : Specifies the name of a table that has a column whose default value is to be modified.
- *column_name* : Specifies the name of a column whose default value is to be modified.
- *value* : Specifies a new default value.

Example

```
;schema a_tbl

=== <Help: Schema of a Class> ===
    a_tbl

<Attributes>
    name                    CHARACTER VARYING(1073741823)
    phone                   CHARACTER VARYING(13) DEFAULT '000-0000-0000'
    age                     INTEGER
    id                      INTEGER AUTO_INCREMENT NOT NULL

<Constraints>
    UNIQUE u_a_tbl_id ON a_tbl (id)

Current transaction has been committed.

ALTER TABLE a_tbl ALTER COLUMN name SET DEFAULT '';
ALTER TABLE a_tbl ALTER COLUMN phone SET DEFAULT '111-1111';

;schema a_tbl

=== <Help: Schema of a Class> ===


 <Class Name>

     a_tbl

 <Attributes>

     name                   CHARACTER VARYING(1073741823) DEFAULT ''
     phone                  CHARACTER VARYING(13) DEFAULT '111-1111'
     age                    INTEGER
     id                     INTEGER AUTO_INCREMENT  NOT NULL

 <Constraints>

     UNIQUE u_a_tbl_id ON a_tbl (id)
```

AUTO_INCREMENT Clause

Description

The **AUTO_INCREMENT** clause can change the initial value of the increment value that is currently defined. However, there should be only one **AUTO_INCREMENT** column defined.

Syntax

```
ALTER TABLE table_name AUTO_INCREMENT = initial_value;
```

- *table_name* : Table name
- *initial_value* : Initial value to alter

Example

```
CREATE TABLE t (i int AUTO_INCREMENT);
ALTER TABLE t AUTO_INCREMENT = 5;

-- when 2 AUTO_INCREMENT constraints are defined on one table, it returns error.
CREATE TABLE t (i int AUTO_INCREMENT, j int AUTO_INCREMENT);
```

```
ALTER TABLE t AUTO_INCREMENT = 5;

ERROR: To avoid ambiguity, the AUTO_INCREMENT table option requires the table to have
exactly one AUTO_INCREMENT column and no seed/increment specification.
```

Caution

You must be careful not to violate constraints (such as a **PRIMARY KEY** or **UNIQUE**) when you alter the initial value of **AUTO_INCREMENT**.

CHANGE/MODIFY Clause

Description

The **CHANGE** clause changes column names or changes the types and the attributes. If the existing column name and a new column name are the same, only the type and the attribute will be changed. The following is an example in which the name of a column is changed using the **CHANGE** clause.

```
CREATE TABLE t1 (a INTEGER);
ALTER TABLE t1 CHANGE a b INTEGER;
```

The following is an example of changing the attribute of a column using the **CHANGE** clause.

```
ALTER TABLE t1 CHANGE a a INTEGER NOT NULL;
```

The **MODIFY** clause can modify the types and the attributes of columns but can not change the names. The following is an example in which the attribute of a column is changed using the **MODIFY** clause.

```
ALTER TABLE t1 MODIFY a INTEGER NOT NULL;
```

If you set the type and the attribute to apply to a new column with the **CHANGE** clause or the **MODIFY** clause, the attribute that is currently defined will not be passed to the attribute of the new column. The following statement changes col1 to the **BIGINT** type but doesn't include the "DEFAULT 1" attribute defined in the existing column.

```
CREATE TABLE t1 (col1 INT DEFAULT 1);
ALTER TABLE t1 MODIFY col1 BIGINT;
```

To include the "DEFAULT 1" attribute in the above example, you should write the following:

```
ALTER TABLE t1 MODIFY col1 BIGINT DEFAULT 1;
```

When you change data types using the **CHANGE** clause or the **MODIFY** clause, the data can be modified. For example, if you decrease the length of a column, the character string can be truncated so you must be careful.

Note that the **CHANGE** syntax used in CUBRID 2008 R3.1 and the earlier versions is no longer supported.

Syntax

```
ALTER TABLE tbl_name table_options;

table_options :
     table_option[, table_option]

table_option :
    CHANGE [COLUMN | CLASS ATTRIBUTE ] old_col_name new_col_name column_definition
             [FIRST | AFTER col_name]
  | MODIFY [COLUMN | CLASS ATTRIBUTE] col_name column_definition
             [FIRST | AFTER col_name]
```

- *tbl_name* : Specifies the name of the table including the column to change.
- *old_col_name* : Specifies the existing column name.
- *new_col_name* : Specifies the column name to change
- *column_definition* : Specifies the type and the attribute of the column to change.
- *col_name* : Specifies the column name to which the type and the attribute of the column to change applies.

Syntax Operation According to Column Attributes

- Type Change : If the value of the system parameter **alter_table_change_type_strict** is set to no, then changing values to other types is allowed, but if it is set to yes then changing is not allowed. The default value of the parameter is **no**. You can change values to all types allowed by the **CAST** function. Changing object types is allowed only by the upper classes (tables) of the objects.
- **NOT NULL**
- If the **NOT NULL** constraint is not specified, it will be removed from a new table even though it is present in the existing table.
- If the **NOT NULL** constraint is specified in the column to change, the result varies depending on the configuration of the system parameter, **alter_table_change_type_strict**.
- If **alter_table_change_type_strict** is set to yes, the column values will be checked. If **NULL** exists, an error will occur, and the change will not be executed.
- If the **alter_table_change_type_strict** is set to no, every existing **NULL** value will be changed to a hard default value of the type to change.
- **DEFAULT** : If the **DEFAULT** attribute is not specified in the column to change, it will be removed from a new table even though it is present in the existing table.
- **AUTO_INCREMENT** : If the **AUTO_INCREMENT** attribute is not specified in the column to change, it will be removed from a new table even though it is present in the existing table.
- **FOREIGN KEY** : You can not change the column with the foreign key constraint that is referred to or refers to.
- Single Column **PRIMARY KEY**
- If the **PRIMARY KEY** constraint is specified in the column to change, a **PRIMARY KEY** is re-created only in which a **PRIMARY KEY** constraint exists in the existing column and the type is upgraded.
- If the **PRIMARY KEY** constraint is specified in the column to change but doesn't exists in the existing column, a **PRIMARY KEY** will be created.
- If a **PRIMARY KEY** constraint exists but is not specified in the column to change, the **PRIMARY KEY** will be maintained.
- Multicolumn **PRIMARY KEY** : If the **PRIMARY KEY** constraint is specified and the type is upgraded, a **PRIMARY KEY** will be re-created.
- Single Column **UNIQUE KEY**
- If the type is upgraded, a **UNIQUE KEY** will be re-created.
- If a **UNIQUE KEY** exists in the existing column and it is not not specified in the column to change, it will be maintained.
- If a **UNIQUE KEY** exists in the existing column to change, it will be created.
- Multicolumn **UNIQUE KEY** : If the column type is changed, an index will be re-created.
- Column with a Non-unique Index : If the column type is changed, an index will be re-created.
- Partition Column: If a table is partitioned by a column, the column can not be changed. Partitions can not be added.
- Column with a Class Hierarchy : You can only change the tables that do not have a lower class. You can not change the lower class that inherits from an uppder class. You can not change the inherited attributes.
- Trigger and View : You must redefine triggers and views directly because they are not changed according to the definition of the column to change.
- Column Sequence : You can change the sequence of columns.
- Name Change : You can change names as long as they do not conflict.

Syntax Operation According to the System Parameter, alter_table_change_type_strict

The **alter_table_change_type_strict** parameter determines whether the value conversion is allowed according to the type change. If the value is no, it can be changed when you change a column type or add a **NOT NULL** constraint. The default value is **no**.

When the value of the parameter, **alter_table_change_type_strict** is no, it will operate depending on the conditions as follows:

- Overflow Occurred while Converting Numbers or Character Strings to Numbers: The minimum value or the maximum value are specified according to the result type conditions, and the warning message will be recorded in the log for the record where overflow has occurred.
- If input values are numbers, their signs will be written to the log.

- If input values are character strings, the signs of the values converted to **DOUBLE** types will be written in the log.
- Character Strings to Convert to Shorter Ones: The record will be updated to the hard default value of the type that is defined and the warning message will be recorded in a log.
- Conversion Failure Due to Other Reasons : The record will be updated to the hard default value of the type that is defined and the warning message will be recorded in a log.

If the value of the **alter_table_change_type_strict** parameter is yes, an error message will be displayed and the changes will be rolled back.

The **ALTER CHANGE** statement checks the possibility of type conversion before updating a record but the type conversion of specific values may fail. For example, if the value format is not correct when you convert **VARCHAR** to **DATE**, the conversion may fail. In this case, the hard default value of the **DATE** type will be assigned.

The hard default value is a value that will be used when you add columns with the **ALTER TABLE â€¦ ADD COLUMN** statement, add or change by converting types with the **ALTER TABLE â€¦ CHANGE/MODIFY** statement. The operation will vary depending on the system parameter, **add_column_update_hard_default** in the **ADD COLUMN** statement.

Hard Default Value by Type

Type	Existence of Hard Default Value	Hard Default Value
INTEGER	Yes	0
FLOAT	Yes	0
DOUBLE	ìœ	0
SMALLINT	Yes	0
DATE	Yes	date'01/01/0001'
TIME	Yes	time'00:00'
DATETIME	Yes	datetime'01/01/0001 00:00'
TIMESTAMP	Yes	timestamp'00:00:00 PM 01/01/1970'
MONETARY	Yes	0
NUMERIC	Yes	0
CHAR	Yes	''
VARCHAR	Yes	''
NCHAR	Yes	N''
VARNCHAR	Yes	N''
SET	Yes	{}
MULTISET	Yes	{}
SEQUENCE	Yes	{}
BIGINT	Yes	0
BIT	Yes	
VARBIT	No	
OBJECT	No	
BLOB	No	
CLOB	No	
ELO	No	

Example 1

```
-- changing the name and position of a column
CREATE TABLE t1(i1 int,i2 int);
INSERT INTO t1 VALUE (1,11),(2,22),(3,33);
SELECT * FROM t1 ORDER BY 1;
            i1           i2
==========================
             1           11
             2           22
             3           33

ALTER TABLE t1 CHANGE i2 i0 INTEGER FIRST;
SELECT * FROM t1 ORDER BY 1;
            i0           i1
==========================
            11            1
            22            2
            33            3
```

Example 2

```
-- adding NOT NULL constraint (strict)
-- alter_table_change_type_strict=yes

CREATE TABLE t1(i int);
INSERT INTO t1 values (11),(NULL),(22);

ALTER TABLE t1 change i i1 integer not null;

In the command from line 1,

ERROR: Cannot add NOT NULL constraint for attribute "i1": there are existing NULL values
for this attribute.
```

Example 3

```
-- adding NOT NULL constraint
-- alter_table_change_type_strict=no

CREATE TABLE t1(i int);
INSERT INTO t1 VALUES (11),(NULL),(22);

ALTER TABLE t1 CHANGE i i1 INTEGER NOT NULL;

SELECT * FROM t1;

           i1
=============
           22
            0
           11
```

Example 4

```
-- change the column's data type (no errors)

CREATE TABLE t1 (i1 int);
INSERT INTO t1 VALUES (1),(-2147483648),(2147483647);

ALTER TABLE t1 CHANGE i1 s1 CHAR(11);

SELECT * FROM t1;

  s1
======================
  '2147483647 '
  '-2147483648'
  '1          '
```

Example 5

```
-- change the column's data type (errors), strict mode
-- alter_table_change_type_strict=yes

CREATE TABLE t1 (i1 int);
INSERT INTO t1 VALUES (1),(-2147483648),(2147483647);

ALTER TABLE t1 CHANGE i1 s1 CHAR(4);

In the command from line 1,

ERROR: ALTER TABLE .. CHANGE : changing to new domain : cast failed, current configuration
doesn't allow truncation or overflow.

-- change the column's data type (errors)
-- alter_table_change_type_strict=no

CREATE TABLE t1 (i1 INT);
INSERT INTO t1 VALUES (1),(-2147483648),(2147483647);

ALTER TABLE t1 CHANGE i1 s1 CHAR(4);

SELECT * FROM t1;

  s1
======================
  '    '
  '    '
  '1   '

-- hard default values have been placed instead of signaling overflow
```

RENAME COLUMN Clause

Description

You can change the name of the column by using the **RENAME COLUMN** clause.

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
RENAME [ COLUMN | ATTRIBUTE ] old_column_name { AS  |  TO  } new_column_name
```

- *table_type* : Specifies the name of a table that has a column to be renamed.
- *old_column_name* : Specifies the name of a column.
- *new_column_name* : Specifies a new column name after the **AS** keyword.

Example

```
ALTER TABLE a_tbl  RENAME COLUMN name AS name1
```

DROP COLUMN Clause

Description

You can delete a column in a table by using the **DROP COLUMN** clause. You can specify multiple columns to delete simultaneously by separating them with commas (,).

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
DROP [ COLUMN | ATTRIBUTE ] column_name, ...
```

- *table_name* : Specifies the name of a table that has a column to be deleted.
- *column_ name* : Specifies the name of a column to be deleted. Multiple columns can be specified by separating them with commas (,).

Example

```
ALTER TABLE a_tbl DROP COLUMN age1,age2,age3;
```

DROP CONSTRAINT Clause

Description

You can drop the constraints pre-defined for the table, such as **UNIQUE**, **PRIMARY KEY** and **FOREIGN KEY** by using the **DROP CONSTRAINT** clause. In this case, you must specify a constraint name. You can check these names by using the CSQL command (**;schema table_name**).

Syntax

```
ALTER [ TABLE | CLASS ] table_name
DROP CONSTRAINT constraint_name
```

- *table_name* : Specifies the name of a table that has a constraint to be dropped.
- *constraint_name* : Specifies the name of a constraint to be dropped.

Example

```
ALTER TABLE a_tbl DROP CONSTRAINT pk_a_tbl_id;
ALTER TABLE a_tbl DROP CONSTRAINT fk_a_tbl_id;
ALTER TABLE a_tbl DROP CONSTRAINT u_a_tbl_id;
```

DROP INDEX Clause

Description

You can delete an index defined for a column by using the **DROP INDEX** clause.

Syntax

```
ALTER [ TABLE | CLASS ] table_name DROP INDEX index_name
```

- *table_name* : Specifies the name of a table that has an index attribute to be deleted.
- *index_name* : Specifies the name of an index to be deleted.

Example

```
ALTER TABLE a_tbl DROP INDEX i_a_tbl_age;
```

DROP PRIMARY KEY Clause

Description

You can delete a primary key constraint defined for a table by using the **DROP PRIMARY KEY** clause. You do have to specify the name of the primary key constraint because only one primary key can be defined by table.

Syntax

```
ALTER [ TABLE | CLASS ] table_name DROP INDEX PRIMARY KEY
```

- *table_name* : Specifies the name of a table that has a primary key constraint to be deleted.

Example

```
ALTER TABLE a_tbl DROP PRIMARY KEY;
```

DROP FOREIGN KEY Clause

Description

You can drop a foreign key constraint defined for a table using the **DROP FOREIGN KEY** clause.

Syntax

```
ALTER [ TABLE | CLASS ] table_name DROP FOREIGN KEY constraint_name
```

- *table_name* : Specifies the name of a table whose constraint is to be deleted.
- *constraint_name* : Specifies the name of foreign key constraint to be deleted.

Example

```
ALTER TABLE a_tbl DROP FOREIGN KEY fk_a_tbl_id;
```

DROP TABLE

Description

You can drop an existing table by the **DROP** statement. Multiple tables can be dropped by a single **DROP** statement. All rows of table are also dropped. If you use it together with the **IF EXISTS** statement, you can prevent errors from occurring and specify multiple tables in one statement.

Syntax

```
DROP [ TABLE | CLASS ] [ IF EXISTS ] <table_specification_comma_list>

<table_specification_comma_list> ::=
<single_table_spec> | ( <table_specification_comma_list> )

<single_table_spec> ::=
|[ ONLY ] table_name
| ALL table_name [ ( EXCEPT table_name, ... ) ]
```

- *table_name* : Specifies the name of the table to be dropped. You can delete multiple tables simultaneously by separating them with commas.
- If a super class name is specified after the **ONLY** keyword, only the super class, not the subclasses inheriting from it, is deleted. If a super class name is specified after the **ALL** keyword, the super classes as well as the subclasses inheriting from it are all deleted. You can specify the list of subclasses not to be deleted after the **EXCEPT** keyword.
- If subclasses that inherit from the super class specified after the **ALL** keyword are specified after the **EXCEPT** keyword, they are not deleted.

Example

```
DROP TABLE history ;

CREATE TABLE t (i INT);

-- DROP TABLE IF EXISTS
DROP TABLE IF EXISTS history, t;
 2 command(s) successfully processed.
 SELECT * FROM t;  In line 1, column 10, ERROR: Unknown class "t".
```

RENAME TABLE

Description

You can change the name of a table by using the **RENAME TABLE** statement and specify a list of the table name to change the names of multiple tables. You can use **TO** instead of **AS**.

Syntax

```
RENAME  [ TABLE | CLASS | VIEW | VCLASS ] old_table_name { AS | TO } new_table_name [,
old_table_name { AS | TO } new_table_name, ... ]
```

- *old_table_name* : Specifies the old table name to be renamed.
- *new_table_name* : Specifies a new table name.

Example

```
RENAME TABLE a_tbl AS aa_tbl;
RENAME TABLE a_tbl TO aa_tbl, b_tbl TO bb_tbl;
```

Caution

The table name can be changed only by the table owner, **DBA** and **DBA** members. The other users must be granted to change the name by the owner or **DBA** (see Granting Authorization for more information on authorization).

Index Definition

CREATE INDEX

Description

Use the **CREATE INDEX** statement to create an index in the specified table.

Syntax

```
CREATE [ REVERSE ] [ UNIQUE ] INDEX [ index_name ]
ON table_name ( column_name[(prefix_length)] [ASC | DESC] [ {, column_name[(prefix_length)]
[ASC | DESC]} ...] ) [ ; ]
```

- **REVERSE** : Creates an index in the reverse order. A reverse index helps to increase sorting speed in descending order.
- **UNIQUE** : Creates an index with unique values.
- *index_name* : Specifies the name of the index to be created. The index name must be unique in the table. If omitted, a name is automatically assigned.
- *prefix_length* : When you specify an index for character- or bit string-type column, you can create an index by specifying the beginning part of the column name as a prefix. You can specify the length of the prefix in bytes in parentheses next to the column name. You cannot specify *prefix_length* in a multiple column index or a **UNIQUE** index. It is impossible to create an index by specifying *prefix_length* as a host variable. If you want to guarantee the query result order in the index in which *prefix_length* is specified, you must specify the **ORDER BY** clause.
- *table_name* : Specifies the name of the table where the index is to be created.
- *column_name* : Specifies the name of the column where the index is to be applied. To create a composite index, specify two or more column names.
- **ASC** | **DESC** : Specifies the sorting order of columns. In case of a **REVERSE** index, **ASC** is ignored and **DESC** is applied.

Example 1

The following is an example of creating a reverse index.

```
CREATE REVERSE INDEX gold_index ON participant(gold);
```

Example 2

The following is an example of creating a multiple column index.

```
CREATE INDEX name_nation_idx ON athlete(name, nation_code);
CREATE INDEX game_date_idx ON game(game_date);
```

Example 3

The following is an example of creating a single column index. In this example, 1-byte long prefix is specified for the nation_code column when creating an index.

```
CREATE INDEX ON game(nation_code(1));
CREATE INDEX game_date_idx ON game(game_date);
```

ALTER INDEX

Description

Use the **ALTER INDEX** statement to rebuild an index. (That is, drop and rebuild an index.) There are the following two ways to specify an index to be rebuilt:

- Specifying it as the name of the index
- Specifying it as the name of the table or the column where the index is specified

Syntax

```
ALTER [ REVERSE ] [ UNIQUE ] INDEX index_name
[ON { ONLY } table_name ( column_name [ {, column_name } ...) ] REBUILD [ ; ]

ALTER [ REVERSE ] [ UNIQUE ] INDEX
ON { ONLY } table_name ( column_name [ {, column_name } ...) REBUILD [ ; ]
```

- **REVERSE** : Creates an index in the reverse order. A reverse index helps to increase sorting speed in descending order.
- **UNIQUE** : Creates an index with unique values.
- *index_name* : Specifies the name of the index to be altered. The index name must be unique in the table.
- *table_name* : Specifies the name of the table where the index is to be created.
- *column_name* : Specifies the name of the column where the index is to be applied. To create a multiple column index, specify two or more column names.

Example

The following are examples of many ways of re-creating indexes:

```
ALTER INDEX i_game_medal ON game(medal) REBUILD;
ALTER INDEX game_date_idx REBUILD;
```

DROP INDEX

Description

Use the **DROP INDEX** statement to drop an index. There are the following two ways to specify the index to be dropped:

- To specify the name of the index
- To specify the name of the table or the column where the index is specified

Syntax

```
DROP [ REVERSE ] [ UNIQUE ] INDEX index_name
[ON table_name ( column_name [ {, column_name } ...) ] [   ]

DROP [ REVERSE ] [ UNIQUE ] INDEX
  ON table_name ( column_name [ {, column_name } ...) [ ]
```

- **REVERSE** : Specifies that the index to be dropped is a reverse index.
- **UNIQUE** : Specifies that the index to be dropped is a unique index.
- *index_name* : Specifies the name of the index to be dropped.
- *table_name* : Specifies the name of the table whose index is to be dropped.
- *column_name* : Specifies the name of the column whose index is to be dropped.

Example

The following are examples of many ways of dropping indexes:

```
DROP INDEX ON game(medal)

DROP INDEX game_date_idx

DROP REVERSE INDEX gold_index ON participant(gold)

DROP INDEX name_nation_idx ON athlete(name, nation_code)
```

VIEW

CREATE VIEW

Overview

Description

A view is a virtual table that does not exist physically. You can create a view by using an existing table or a query. **VIEW** and **VCLASS** are used interchangeably.

Use **CREATE VIEW** statement to create a view.

Syntax

```
CREATE [OR REPLACE] {VIEW | VCLASS} <view_name>
                           [ <subclass_definition> ]
                           [ ( <view_column_def_comma_list> ) ]
                           [ CLASS ATTRIBUTE
                             ( <column_definition_comma_list> ) ]
                           [ METHOD <method_definition_comma_list> ]
                           [ FILE <method_file_comma_list> ]
                           [ INHERIT <resolution_comma_list> ]
                           [ AS <select_statement> ]
                           [ WITH CHECK OPTION ]

<view_column_definition> ::= <column_definition> | <column_name>

<column_definition> :
column_name column_type [ <default_or_shared> ] [ <column_constraint_list>]

<default_or_shared> :
{SHARED [ <value_specification> ] | DEFAULT <value_specification> } |
AUTO_INCREMENT [ (seed, increment) ]

<column_constraint> :
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY REFERENCES...

<subclass_definition> :
{ UNDER | AS SUBCLASS OF } table_name_comma_list

<method_definition> :
[ CLASS ] method_name
[ ( [ argument_type_comma_list ] ) ]
[ result_type ]
[ FUNCTION function_name ]

<resolution> :
[ CLASS ] { column_name | method_name } OF superclass_name
[ AS alias ]
```

- **OR REPLACE** : If the keyword **OR REPLACE** is specified after **CREATE**, the existing virtual table is replaced by a new one without displaying any error message, even when the view_name overlaps with the existing virtual table name.
- *view_name* : Specify the name of the table to be created. Must be unique in the database.
- *view_column_definition*
- *column_name* : Defines a column of the virtual table.
- *column_type* : Specifies the data type of the column.

AS *select_statement* : A valid **SELECT** statement must be specified. A virtual table is created on this basis.

WITH CHECK OPTION : If this option is specified, the update or insert operation is possible only when the condition specified in the **WHERE** clause of the *select_statement* is satisfied. Therefore, this option is used to disallow the update of a virtual table that violates the condition.

Example

```
CREATE TABLE a_tbl(
id INT NOT NULL,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333'), (4, NULL), (5,
NULL);


--creating a new view based on AS select_statement from a_tbl
CREATE VIEW b_view AS SELECT * FROM a_tbl WHERE phone IS NOT NULL WITH CHECK OPTION;
SELECT * FROM b_view;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'

--WITH CHECK OPTION doesn’t allow to update column value which violates WHERE clause
UPDATE b_view SET phone=NULL;

In line 1, column 72,

ERROR: Check option exception on view b_view.


--creating view which name is as same as existing view name
CREATE OR REPLACE VIEW b_view AS SELECT * FROM a_tbl ORDER BY id DESC;

--the existing view has been replaced as a new view by OR REPLACE keyword
SELECT * FROM b_view;

           id  phone
===================================
            5  NULL
            4  NULL
            3  '333-3333'
            2  '222-2222'
            1  '111-1111'
```

Condition for Creating Updatable VIEW

Description

To update data in a virtual table, it must be updatable because an option is needed to define data.

A virtual table is updatable if it satisfies the following conditions:

- The **FROM** clause must include only one table or updatable virtual table. However, two tables included in parentheses as in **FROM** (class_x, class_y) can be updated because they represent one table.
- The **DISTINCT** or **UNIQUE** statement must not be included.
- The **GROUP BY... HAVING** statement must not be included.
- Aggregate functions such as **SUM**( ) or **AVG**( ) must not be included.
- The entire query must consist of queries that can be updated by **UNION ALL**, not by **UNION**. However, the table must exist only in one of the queries that constitute **UNION ALL**.
- If an row is inserted into a virtual table created by using the **UNION ALL** statement, the system determines which table the row will be inserted into. This cannot be done by the user. To control this, the user must manually insert the row or create a separate virtual table for insertion.

Even when all rules above are satisfied, each column of the updatable virtual table may not be updatable. For a column to be updatable, the following rules must be observed:

- Path expressions must not be updatable.
- Columns of number type with an arithmetic operator must not be updatable.

Even though the column defined in the virtual table is updatable, the virtual table can be updated only when there is an appropriate update privilege granted on the table included in the **FROM** clause. Also, there must be an access privilege on the virtual table. The way to grant an access privilege on a virtual table is the same as on a table. For more information on granting authorizations, see the Granting Authorization section.

ALTER VIEW

ADD QUERY Clause

Description

You can add a new query to a query specification by using the **ADD QUERY** clause of the **ALTER VIEW** statement. 1 is assigned to the query defined when a virtual table was created, and 2 is assigned to the query added by the **ADD QUERY** clause.

Syntax

```
ALTER [ VIEW | VCLASS ] view_name
ADD QUERY select_statement
[ INHERIT resolution [ {, resolution }_ ] ]

resolution :
{ column_name | method_name } OF super class_name [ AS alias ]
```

- *view_name* : Specifies the name of the virtual table where the query is to be added.
- *select_statement* : Specifies the query to be added.

Example

```
SELECT * FROM b_view;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            4  NULL
            5  NULL


ALTER VIEW b_view ADD QUERY SELECT * FROM a_tbl WHERE id IN (1,2);
SELECT * FROM b_view;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            4  NULL
            5  NULL
            1  '111-1111'
            2  '222-2222'
```

AS SELECT Clause

Description

You can change the **SELECT** query defined in the virtual table by using the **AS SELECT** clause in the **ALTER VIEW** statement. This function is working like the **CREATE OR REPLACE** statement. You can also change the query by specifying the query number 1 in the **CHANGE QUERY** clause of the **ALTER VIEW** statement.

Syntax

```
ALTER [ VIEW | VCLASS ] view_name AS select_statement
```

- *view_name* : Specifies the name of the virtual table to be modified.

- *select_statement* : Specifies the new query statement to replace the **SELECT** statement defined when the virtual table is created.

Example

```
ALTER VIEW b_view AS SELECT * FROM a_tbl WHERE phone IS NOT NULL;
SELECT * FROM b_view;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
```

CHANGE QUERY Clause

Description

You can change the query defined in the query specification by using the **CHANGE QUERY** clause reserved word of the **ALTER VIEW** statement.

Syntax

```
ALTER [ VIEW | VCLASS ] view_name
    CHANGE QUERY [ integer ] select_statement [ ; ]
```

- *view_name* : Specifies the name of the virtual table to be changed.
- *integer* : Specifies the number value of the query to be changed. The default value is 1.
- *select_statement* : Specifies the new query that will replace the query whose query number is *integer*.

Example

```
--adding select_statement which query number is 2 and 3 for each
ALTER VIEW b_view ADD QUERY SELECT * FROM a_tbl WHERE id IN (1,2);
ALTER VIEW b_view ADD QUERY SELECT * FROM a_tbl WHERE id = 3;
SELECT * FROM b_view;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            4  NULL
            5  NULL
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'

--altering view changing query number 2
ALTER VIEW b_view CHANGE QUERY 2 SELECT * FROM a_tbl WHERE phone IS NULL;
SELECT * FROM b_view;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            4  NULL
            5  NULL
            4  NULL
            5  NULL
            3  '333-3333'
```

DROP QUERY Clause

Description

You can drop a query defined in the query specification by using the **DROP QUERY** of the **ALTER VIEW** statement.

Example

```
ALTER VIEW b_view DROP QUERY 2,3;
SELECT * FROM b_view;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            4  NULL
            5  NULL


5 rows selected.
```

DROP VIEW

Description

You can drop a view by using the **DROP VIEW** clause. The way to drop a view is the same as to drop a regular table.

Syntax

```
DROP [ VIEW | VCLASS ] view_name [ { ,view_name , ... } ]
```

- *view_name* : Specifies the name of the virtual table to be dropped.

Example

```
DROP VIEW b_view;
```

RENAME VIEW

Description

You can change the name of a virtual table by using the **RENAME VIEW** statement.

Syntax

```
RENAME [ TABLE |CLASS | VIEW | VCLASS ] old_view_name AS new_view_name [ ; ]
```

- *old_view_name* : Specifies the name of the table to be modified.
- *new_view_name* : Specifies the new name of the virtual table.

Example

The following is an example of renaming a view name to game_2004.

```
RENAME VIEW game_2004 AS info_2004;
```

SERIAL

CREATE SERIAL

Serial is an object that creates a unique sequence number, and has the following characteristics.

- The serial is useful in creating a unique sequence number in multi-user environment.
- Generated serial numbers are not related with table so, you can use the same serial in multiple tables.
- All users including **public** can create a serial object. Once it is created, all users can get the number by using **CURRENT_VALUE** and **NEXT_VALUE**.
- Only owner of a created serial object and **dba** can update or delete a serial object. If an owner is **public**, all users can update or delete it.

Description

You can create a serial object in the database by using the **CREATE SERIAL** statement.

Syntax

```
CREATE SERIAL serial_name
[ START WITH initial ]
[ INCREMENT BY interval]
[ MINVALUE min | NOMINVALUE ]
[ MAXVALUE max | NOMAXVALUE ]
[ CACHE integer | NOCACHE ]
```

- *serial_identifier* : Specifies the name of the serial to be generated.
- **START WITH** *initial* : Specifies the initial value of serial with 38 digits or less. In the ascending serial, that is its minimum value. In the descending serial, this is its maximum value.
- **INCREMENT BY** *interval* : Specifies the increment of the serial. You can specify any integer with 38 digits or less except for zero at *interval*. The absolute value of the *interval* must be smaller than the difference between **MAXVALUE** and **MINVALUE**. If a negative number is specified, the serial is in descending order otherwise, it is in ascending order. The default value is **1**.
- **MINVALUE** : Specifies the minimum value of the serial, with 38 digits or less. **MINVALUE** must be smaller than or equal to the initial value and smaller than the maximum value.
- **NOMINVALUE** : 1 is set automatically as a minimum value for the ascending serial $-(10)^{38}$ for the descending serial.
- **MAXVALUE** : Specifies the maximum number of the serial with 38 digits or less. **MAXVALUE** must be smaller than or equal to the initial value and greater than the minimum value.
- **NOMAXVALUE** : $(10)^{37}$ is set automatically as a maximum value for the ascending serial -1 for the descending serial.
- **CYCLE** : Specifies that the serial will be generated continuously after reaching the maximum or minimum value. When a serial in ascending order reaches the maximum value, the minimum value is created as the next value; when a serial in descending order reaches the minimum value, the maximum value is created as the next value.
- **NOCYCLE** : Specifies that the serial will not be generated any more after reaching the maximum or minimum value. The default value is **NOCYCLE**.
- **CACHE** : Saves as many serials as the number specified by "integer" in the cache to improve the performance of the serials and fetches a serial value when one is requested. If all cached values are used up, as many serials as "integer" are fetched again from the disk to the memory. If the database server stops accidently, all cached serial values are deleted. For this reason, the serial values before and after the restart of the database server may be discontinuous. Because the transaction rollback dose not affect the cached serial values, the request for the next serial will return the next value of the value used (or fetched) lastly when the transaction is rolled back. The "integer" after the **CACHE** keyword cannot be omitted. If the "integer" is equal to or smaller than 1, the serial cache is not applied.
- **NOCACHE** : Does not use the serial cache feature. The serial value is updated every time and a new serial value is fetched from the disk upon each request.

Example 1

```
--creating serial with default values
CREATE SERIAL order_no;

--creating serial within a specific range
CREATE SERIAL order_no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000;
--creating serial with specifying the number of cached serial values
CREATE SERIAL order_no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000 CACHE 3;

--selecting serial information from the db_serial class
SELECT * FROM db_serial;

  name            current_val      increment_val         max_val         min_val         c
yclic      started       cached_num        att_name
==========================================================================================
==========================================================
'order_no'      10006            2                    20000           10000
  0            1               3            NULL
```

Example 2

The following is an example of creating the athlete_idx table to save athlete codes and names and then creating an instance by using the *order_no*. NEXT_VALUE increases the serial number and returns its value.

```
CREATE TABLE athlete_idx( code INT, name VARCHAR(40) );
CREATE SERIAL order_no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000;
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Park');
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Kim');
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Choo');
INSERT INTO athlete_idx VALUES (order_no.CURRENT_VALUE, 'Lee');
SELECT * FROM athlete_idx;

         code  name
===================================
        10000  'Park'
        10002  'Kim'
        10004  'Choo'
        10004  'Lee'
```

ALTER SERIAL

Description

With the **ALTER SERIAL** statement, you can update the increment of the serial value, set or delete its initial or minimum/maximum values, and set its cycle attribute.

Syntax

```
ALTER SERIAL serial_identifier
[ INCREMENT BY interval ]
[ START WITH initial_value ]
[ MINVALUE min | NOMINVALUE ]
[ MAXVALUE max | NOMAXVALUE ]
[ CACHE integer | NOCACHE ]
```

- *serial_identifier* : Specifies the name of the serial to be created.
- **INCREMENT BY** *interval* : Specifies the increment of the serial. For the *interval*, you can specify any integer with 38 digits or less except for zero. The absolute value of the *interval* must be smaller than the difference between **MAXVALUE** and **MINVALUE**. If a negative number is specified, the serial is in descending order; otherwise, it is in ascending order. The default value is **1**.
- **START WITH** *initial_value* : Changes the initial value of Serial.
- **MINVALUE** : Specifies the minimum value of the serial with 38 digits or less. **MINVALUE** must be smaller than or equal to the initial value and smaller than the maximum value.
- **NOMINVALUE** : 1 is set automatically as a minimum value for the ascending serial; $-(10)^{36}$ for the descending serial.

- **MAXVALUE** : Specifies the maximum number of the serial with 38 digits or less. **MAXVALUE** must be smaller than or equal to the initial value and greater than the minimum value.
- **NOMAXVALUE** : $(10)^{37}$ is set automatically as a maximum value for the ascending serial; -1 for the descending serial.
- **CYCLE** : Specifies that the serial will be generated continuously after reaching the maximum or minimum value. If the ascending serial reaches the maximum value, the minimum value is generated as the next value. If the descending serial reaches the minimum value, the maximum value is generated as the next value.
- **NOCYCLE** : Specifies that the serial will not be generated any more after reaching the maximum or minimum value. The default is **NOCYCLE**.
- **CACHE** : Saves as many serials as the number specified by integer in the cache to improve the performance of the serials and fetches a serial value when one is requested. The "integer" after the **CACHE** keyword cannot be omitted. If a number equal to or smaller than 1 is specified, the serial cache is not applied.
- **NOCACHE** : It does not use the serial cache feature. The serial value is updated every time and a new serial value is fetched from the disk upon each request.

Caution In CUBRID 2008 R1.x version, the serial value can be modified by updating the db_serial talbe, a system catalog. However, in CUBRID 2008 R2.0 version or above, the modification of the db_serial table is not allowed but use of the **ALTER SERIAL** statement is allowed. Therefore,if an **ALTER SERIAL** statement is included in the data exported (unloaddb) from CUBRID 2008 R2.0 or above, it is not allowed to import (loaddb) the data in CUBRID 2008 R1.x or below.

Example

```
--altering serial by changing start and incremental values
ALTER SERIAL order_no START WITH 100 INCREMENT BY 2;

--altering serial to operate in cache mode
ALTER SERIAL order_no CACHE 5;

--altering serial to operate in common mode
ALTER SERIAL order_no NOCACHE;
```

DROP SERIAL

Description

With the **DROP SERIAL** statement, you can drop a serial object from the database.

Syntax

```
DROP SERIAL serial_identifier
```

- *serial_identifier* : Specifies the name of the serial to be dropped.

Example

The following is an example of dropping the *order_no* serial.

```
DROP SERIAL order_no;
```

Use SERIAL

Description

You can access and update a serial by serial name and a reserved word pair.

Syntax

```
serial_identifier.CURRENT_VALUE
serial_identifier.NEXT_VALUE
```

- *serial_identifier*.**CURRENT_VALUE** : Returns the current serial value.
- *serial_identifier*.**NEXT_VALUE** : Increments the serial value and returns the result.

Example

The following is an example to create a table athlete_idx where athlete numbers and names are stored and to create the instances by using a serial order_no.

```
CREATE TABLE athlete_idx( code INT, name VARCHAR(40) );
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Park');
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Kim');
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Choo');
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Lee');SELECT * FROM athlete_idx;

         code  name
===================================
        10000  'Park'
        10002  'Kim'
        10004  'Choo'
        10006  'Lee'
```

Caution

- When you use a serial for the first time after creating it, **NEXT_VALUE** returns the initial value. Subsequently, the sum of the current value and the increment are returned.

Operators and Functions

Logical Operators

Description

For logical operators, boolean expressions or expressions that evaluates to an integer value are specified as operands; **TRUE**, **FALSE** or **NULL** is returned as the result. If the INTEGER value is used, 0 is evaluated to **FALSE** and the other values are evaluated to **TRUE**. If a boolean value is used, 1 is evaluated to **TRUE** and 0 is evaluated to **FALSE**.

The following table shows the logic operators supported by CUBRID.

Logical Operators Supported by CUBRID

Logical Operator	Description	Condition
AND, **&&**	If all operands are **TRUE**, it returns **TRUE**.	a **AND** b
OR, ||	If none of operands is **NULL** and one or more operand is **TRUE**, it returns **TRUE**. If **pipes_as_concat** is no that is a parameter related to SQL statements, a double pipe symbol can be used as **OR** operator.	a **OR** b
XOR	If none of operand is **NULL** and each of operand has a different value, it returns **TRUE**.	a **XOR** b
NOT, **!**	A unary operator. If a operand is **FALSE**, it returns **TRUE**. If it is **TRUE**, returns **FALSE**.	**NOT** a

True Table of Logical Operators

a	b	a AND b	a OR b	NOT a	a XOR b
TRUE	**TRUE**	TRUE	TRUE	FALSE	FALSE
TRUE	**FALSE**	FALSE	TRUE	FALSE	TRUE
TRUE	**NULL**	NULL	TRUE	FALSE	NULL
FALSE	**FALSE**	FALSE	FALSE	TRUE	FALSE
FALSE	**NULL**	FALSE	NULL	TRUE	NULL

Note

You should put the logical expressions in brackets in the **SELECT** list.

```
SELECT 1 = 1 FROM db_root;
ERROR: syntax error, unexpected '='

SELECT (1 = 1) FROM db_root;
        (1=1)
=============
            1
```

Comparison Operators

Description

The comparison operator compares the operand on the left and on the right, and returns 1 or 0. Operands of the comparison operation must be of the same data type. Therefore, implicit type casting by the system or implicit type casting by the user is required.

The following table shows the comparison operators supported by CUBRID and their return values.

Comparison Operators Supported by CUBRID

Comparison Operator	Description	Predicate	Return Value
=	A general equal sign. It compares whether the values of the left and right operands are the same. Returns **NULL** if one or more operand is NULL.	1=>2 1=NULL	0 NULL
<=>	A NULL-safe equal sign. It compares whether the values of the left and right operands are the same including **NULL**. Returns 1 if both operands are **NULL**.	1<==>2 1<=> NULL	0 0
<>, !=	The value of left operand is not equal to that of right operand. If any operand value is **NULL**, **NULL** is returned.	1<>2	1
>	The value of left operand is greater than that of right operand. If any operand value is **NULL**, **NULL** is returned.	1>2	0
<	The value of left operand is less than that of right operand. If any operand value is **NULL**, **NULL** is returned.	1<2	1
>=	The value of left operand is greater than or equal to that of right operand. If any operand value is **NULL**, **NULL** is returned.	1>=2	0
<=	The value of left operand is less than or equal to that of right operand. If any operand value is **NULL**, **NULL** is returned.	1<=2	1
IS *boolean_value*	Compares whether the value of the left operand is the same as boolean value of the right. The boolean value may be **TRUE**, **FALSE** (or **NULL**).	1 IS FALSE	0
IN NOT *boolean_value*	Compares whether the value of the left operand is the same as boolean value of the right. The boolean value may be **TRUE**, **FALSE** (or **NULL**).	1 IS NOT FALSE	1

Syntax 1

```
expression  comparison_operator  expression

expression :
• bit string
• character string
• numeric value
• date-time value
• collection value
• NULL

comparison_operator :
=
| <=>
| <>
| !=
| >
| <
| >=
| <=
```

Syntax 2

```
expression IS [NOT] boolean_value

expression :
• bit string
• character string
• numeric value
• date-time value
```

```
• collection value
• NULL

boolean_value :
< UNKNOWN | NULL>
| TRUE
| FALSE
```

- *expression* : Declares an expression to be compared.
- *bit string* : A Boolean operation can be performed on bit strings, and all comparison operators can be used for comparison between bit strings. If you compare two expressions with different lengths, 0s are padded at the end of the shorter one.
- *character string* : When compared by a comparison operator, two character strings must have the same character sets. The comparison is determined by the collation sequence of the character code set. If you compare two character strings with different lengths, blanks are padded at the end of the shorter one before comparison so that they have the same length.
- *numeric value* : The Boolean operator can be performed for all numeric values and any types of comparison operator can be used. When two different numeric types are compared, the system implicitly performs type casting. For example, when an **INTEGER** value is compared with a **DECIMAL** value, the system first casts **INTEGER** to **DECIMAL** before it performs comparison. When you compare a **FLOAT** value, you must specify the range instead of an exact value because the processing of **FLOAT** is dependent on the system.
- *date-time value* : If two date-time values with the same type are compared, the order is determined in time order. That is, when comparing two date-time values, the earlier date is considered to be smaller than the later date. You cannot compare date-time values with different type by using a comparison operator; therefore, you must explicitly convert it. However, comparison operation can be performed between DATE, TIMESTAMP, and DATETIME because they are implicitly converted.
- *collection value* : When comparing two sequences each element of the two sequences is compared in the order that is specified at the time of sequence creation. Comparison between sets or multisets is overloaded by an appropriate operator. You can perform comparison operations on sets, multisets, lists or sequence sets by using a containment operator explained later in this chapter. For more information, see Containment Operators.
- **NULL** : The **NULL** value is not included in the value range of any data type. Therefore, comparison between **NULL** values is only allowed to determine if the given value is **NULL** or not. An implicit type cast does not take place when a **NULL** value is assigned to a different data type. For example, when an attribute of **INTEGER** type has a **NULL** and is compared with a floating point type, the **NULL** value is not coerced to **FLOAT** before comparison is made. A comparison operation on the **NULL** value does not return a result.

Example

```
EVALUATE (1 <> 0); -- 1 is outputted because it is TRUE.
EVALUATE (1 != 0); -- 1 is outputted because it is TRUE.
EVALUATE (0.01 = '0.01'); -- An error occurs because a numeric data type is compared with
a character string type.
EVALUATE (1 = NULL); -- NULL is outputted.
EVALUATE (1 <=> NULL); -- 0 is outputted because it is FALSE.
EVALUATE (1.000 = 1); -- 1 is outputted because it is TRUE.
EVALUATE ('cubrid' = 'CUBRID'); -- 0 is outputted because it is case sensitive.
EVALUATE ('cubrid' = 'cubrid'); -- 1 is outputted because it is TRUE.
EVALUATE (SYSTIMESTAMP = CAST(SYSDATETIME AS TIMESTAMP)); -- 1 is outputted after casting
the type explicitly and then performing comparison operator.
EVALUATE (SYSTIMESTAMP = SYSDATETIME)); 0 is outputted after casting the type implicitly
and then performing comparison operator.
EVALUATE (SYSTIMESTAMP <> NULL); -- NULL is returned without performing comparison
operator.
EVALUATE (SYSTIMESTAMP IS NOT NULL); -- 1 is returned because it is no NULL.
```

Arithmetic Operators

Arithmetic Operators

Description

For arithmetic operators, there are binary operators for addition, subtraction, multiplication, or division, and unary operators to represent whether the number is positive or negative. The unary operators to represent the numbers' positive/negative status have higher priority over the binary operators.

Arithmetic Operators Supported by CUBRID

Arithmetic Operator	Description	Operator	Return Value
+	Addition	1+2	3
-	Subtraction	1-2	-1
*****	Multiplication	1*2	2
/	Division. Returns quotient.	1/2.0	0.500000000
DIV	Division. Returns quotient.	1 DIV 2	0
%, **MOD**	Division. Returns quotient. An operator must be an integer type, and it always returns integer. If an operand is real number, the **MOD** function can be used.	1 % 2 1 MOD 2	1

Syntax

```
expression  mathematical_operator  expression
expression :
• bit string
• character string
• numeric value
• date-time value
• collection value
• NULL

mathematical_operator :
• set_arithmetic_operator
• arithmetic_operator

arithmetic_operator :
• +
• -
• *
• /, DIV
• %, MOD

set_arithmetic_operator :
• UNION                   (Union)
• DIFFERENCE          (Difference)
• INTERSECT | INTERSECTION   (Intersection)
```

- *expression* : Declares the mathematical operation to be calculated.
- *mathematical_operator* : A operator that performs an operation the arithmetic and the set operators are applicable.
- *set_arithmetic_operator* : A set arithmetic operator that performs operations such as union, difference and intersection on collection type operands.
- *arithmetic_operator* : An operator to perform the four fundamental arithmetic operations.

Arithmetic Operations and Type Casting of Numeric Data Types

Description

All numeric data types can be used for arithmetic operations. The result type of the operation differs depending on the data types of the operands and the type of the operation. The following table shows the result data types of addition/subtraction/multiplication for each operand type.

Result Data Type by Operand Type

	INT	NUMERIC	FLOAT	DOUBLE	MONETARY
INT	**INT** (**BIGINT**)	**NUMERIC**	**FLOAT**	**DOUBLE**	**MONETARY**
NUMERIC	**NUMERIC**	**NUMERIC** (*p* and *s* are also converted)	**DOUBLE**	**DOUBLE**	**MONETARY**

FLOAT	**FLOAT**	**DOUBLE**	**FLOAT**	**DOUBLE**	**MONETARY**
DOUBLE	**DOUBLE**	**DOUBLE**	**DOUBLE**	**DOUBLE**	**MONETARY**
MONETARY	**MONETARY**	**MONETARY**	**MONETARY**	**MONETARY**	**MONETARY**

Note that the result type of the operation does not change if all operands are of the same data type but type casting occurs exceptionally in division operations. An error occurs when a denominator, i.e. a divisor, is 0.

If one of the operands is a **MONETARY** type, all operation results are cast to **MONETARY** type because a **MONETARY** type uses the same operation methods as the DOUBLE type.

The following table shows the total number of digits (*p*) and the number of digits after the decimal point (*s*) of the operation results when all operands are of the **NUMERIC** type.

Result of NUMERIC Type Operation

Operation	**Maximum Precision**	**Maximum Scale**
N(p1, s1) + N(p2, s2)	max(p1-s1, p2-s2)+max(s1, s2) +1	max(s1, s2)
N(p1, s1) - N(p2, s2)	max(p1-s1, p2-s2)+max(s1, s2)	max(s1, s2)
N(p1, s1) * N(p2, s2)	p1+p2+1	s1+s2
N(p1, s1) / N(p2, s2)	Let Pt = p1+max(s1, s2) + s2 - s1 when s2 > 0 and Pt = p1 in other cases; St = s1 when s1 > s2 and s2 in other cases; the number of decimal places is min(9-St, 38-Pt) + St when St < 9 and St in other cases.	

Example

```
--int * int
SELECT 123*123;
      123*123
=============
        15129

-- int * int returns overflow error
SELECT (1234567890123*1234567890123);

ERROR: Data overflow on data type bigint.

-- int * numeric returns numeric type
SELECT (1234567890123*CAST(1234567890123 AS NUMERIC(15,2)));
 (1234567890123* cast(1234567890123 as numeric(15,2)))
======================
  1524157875322755800955129.00

-- int * float returns float type
SELECT (1234567890123*CAST(1234567890123 AS FLOAT));
 (1234567890123* cast(1234567890123 as float))
===============================================
                                  1.524158e+024

-- int * double returns double type
SELECT (1234567890123*CAST(1234567890123 AS DOUBLE));
 (1234567890123* cast(1234567890123 as double))
================================================
                          1.524157875322756e+024

-- numeric * numeric returns numeric type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS NUMERIC(15,2)));
 ( cast(1234567890123 as numeric(15,2))* cast(1234567890123 as numeric(15,2)))
======================
  1524157875322755800955129.0000

-- numeric * float returns double type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS FLOAT));
 ( cast(1234567890123 as numeric(15,2))* cast(1234567890123 as float))
=======================================================================
                                                 1.524157954716582e+024
```

```
-- numeric * double returns double type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS DOUBLE));
 ( cast(1234567890123 as numeric(15,2))* cast(1234567890123 as double))
=========================================================================
                                                  1.524157875322756e+024

-- float * float returns float type
SELECT (CAST(1234567890123 AS FLOAT)*CAST(1234567890123 AS FLOAT));
 ( cast(1234567890123 as float)* cast(1234567890123 as float))
================================================================
                                                  1.524158e+024
-- float * double returns float type
SELECT (CAST(1234567890123 AS FLOAT)*CAST(1234567890123 AS DOUBLE));
 ( cast(1234567890123 as float)* cast(1234567890123 as double))
=================================================================
                                          1.524157954716582e+024

-- double * double returns float type
SELECT (CAST(1234567890123 AS DOUBLE)*CAST(1234567890123 AS DOUBLE));
 ( cast(1234567890123 as double)* cast(1234567890123 as double))
==================================================================
                                           1.524157875322756e+024

-- int / int returns int type without type conversion or rounding
SELECT 100100/100000;
  100100/100000
===============
              1

-- int / int returns int type without type conversion or rounding
SELECT 100100/200200;
  100100/200200
===============
              0

-- int / zero returns error
SELECT 100100/(100100-100100);
ERROR: Attempt to divide by zero.
```

Arithmetic Operations and Type Casting of DATE/TIME Data Types

Description

If all operands are date/time type, only a subtraction operation is allowed and its return value is **INT**. Note that the unit of the operation differs depending on the types of the operands. Both addition and subtraction operations are allowed in case of date/time and integer types In this case, operation units and return values are date/time data type.

The following table shows operations allowed for each operand type, and their result types.

Allowable Operation and Result Data Type by Operand Type

	TIME (in seconds)	**DATE (in day)**	**TIMESTAMP (in seconds)**	**DATETIME (in milliseconds)**	**INT**
TIME	A subtraction is allowed. **INT**	X	X	X	An addition and a subtraction are allowed. **INT**
DATE	X	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	An addition and a subtraction are allowed. **DATE**
TIMESTAMP	X	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	An addition and a subtraction are allowed. **TIMESTAMP**
DATETIME	X	A subtraction	A subtraction is	A subtraction is	An addition and

		is allowed. **INT**	allowed. **INT**	allowed. **INT**	a subtraction are allowed. **DATETIME**
INT	An addition and a subtraction are allowed. **TIME**	An addition and a subtraction are allowed. **DATE**	An addition and a subtraction are allowed. **TIMESTAMP**	An addition and a subtraction are allowed. **DATETIME**	All operations are allowed.

Remark

If any of the date/time arguments contains **NULL**, **NULL** is returned.

Example

```
-- initial systimestamp value
SELECT SYSDATETIME;
  SYSDATETIME
===============================
  07:09:52.115 PM 01/14/2010

-- time type + 10(seconds) returns time type
SELECT (CAST (SYSDATETIME AS TIME) + 10);
 ( cast( SYS_DATETIME  as time)+10)
====================================
  07:10:02 PM

-- date type + 10 (days) returns date type
SELECT (CAST (SYSDATETIME AS DATE) + 10);
 ( cast( SYS_DATETIME  as date)+10)
====================================
  01/24/2010

-- timestamp type + 10(seconds) returns timestamp type
SELECT (CAST (SYSDATETIME AS TIMESTAMP) + 10);
 ( cast( SYS_DATETIME  as timestamp)+10)
=========================================
  07:10:02 PM 01/14/2010

-- systimestamp type + 10(milliseconds) returns systimestamp type
SELECT (SYSDATETIME  + 10);
 ( SYS_DATETIME +10)
===============================
  07:09:52.125 PM 01/14/2010

SELECT DATETIME '09/01/2009 03:30:30.001 pm'- TIMESTAMP '08/31/2009 03:30:30 pm';
 datetime '09/01/2009 03:30:30.001 pm'-timestamp '08/31/2009 03:30:30 pm'
=======================================
  86400001

SELECT TIMESTAMP '09/01/2009 03:30:30 pm'- TIMESTAMP '08/31/2009 03:30:30 pm';
 timestamp '09/01/2009 03:30:30 pm'-timestamp '08/31/2009 03:30:30 pm'
=======================================
  86400
```

Set Operators

Set Arithmetic Operators

Set Arithmetic Operators

To evaluate set operations such as union, difference or intersection for **SET**, **MULTISET** or **LIST (SEQUENCE)** types, you can use +, - or * operators respectively.

The following table shows a summary of how to use these operators.

Result Data Type by Operand Type

	SET	MULTISET	LIST (=SEQUENCE)
SET	+, -, * : **SET**	+, -, * : **MULTISET**	+, -, * : **MULTISET**
MULTISET	+, -, * : **MULTISET**	+, -, * : **MULTISET**	+, -, * : **MULTISET**
LIST (=SEQUENCE)	+ : **MULTISET** - : **MULTISET** * : **MULTISET**	+ : **MULTISET** - : **MULTISET** * : **MULTISET**	+ : **LIST** - : **MULTISET** * : **MULTISET**

Syntax

```
value_expression set_arithmetic_operator value_expression

value_expression :
• collection value
• NULL

set_arithmetic_operator :
• + (union)
• - (difference)
• * (intersection)
```

Example

```
SELECT ((CAST ({3,3,3,2,2,1} AS SET))+(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as set))+( cast({4, 3, 3, 2} as multiset)))
======================
  {1, 2, 2, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISET))+(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as multiset))+( cast({4, 3, 3, 2} as multiset)))
======================
  {1, 2, 2, 2, 3, 3, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))+(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as sequence))+( cast({4, 3, 3, 2} as multiset)))
======================
  {1, 2, 2, 2, 3, 3, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS SET))-(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as set))-( cast({4, 3, 3, 2} as multiset)))
======================
  {1}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISET))-(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as multiset))-( cast({4, 3, 3, 2} as multiset)))
======================
  {1, 2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))-(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as sequence))-( cast({4, 3, 3, 2} as multiset)))
======================
  {1, 2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS SET))*(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as set))*( cast({4, 3, 3, 2} as multiset)))
======================
  {2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISET))*(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as multiset))*( cast({4, 3, 3, 2} as multiset)))
======================
  {2, 3, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))*(CAST ({4,3,3,2} AS MULTISET)));
 (( cast({3, 3, 3, 2, 2, 1} as sequence))*( cast({4, 3, 3, 2} as multiset)))
======================
{2, 3, 3}
```

Assigning Collection Value to Variable

For a collection value to be assigned to a variable, the outer query must return a single row as the result. The following is an example of assigning a collection value to a variable. The outer query must return only a single row as follows:

```
SELECT SET(SELECT name
FROM people
WHERE ssn in {'1234', '5678'})
TO :"names"
FROM TABLE people;
```

Statement Set Operators

Description

Statement set operators are used to get union, difference or intersection on the result of more than one query statement specified as an operand. Note that the data types of the data to be retrieved from the target tables of the two query statements must be identical or implicitly castable.

The following table shows statement set operators supported by CUBRID and their examples.

Statement Set Operators Supported by CUBRID

Statement Set Operator	Description	Note
UNION	Union Duplicates are not allowed.	Outputs all instance results containing duplicates with **UNION ALL**
DIFFERENCE	Difference Duplicates are not allowed.	Same as the **EXCEPT** operator Outputs all instance results containing duplicates with **DIFFERENCE ALL**
INTERSECTION	Intersection Duplicates are not allowed.	Same as the **INTERSECTION** operator Outputs all instance results containing duplicates with **INTERSECTION ALL**

Syntax

```
query_term statement_set_operator[qualifier] query_term
[{statement_set_operator[qualifier] query_term}];

query_term :
• query_specification
• subquery

qualifier :
• DISTINCT or DISTINCTROW (A returned instance is a distinct value.)
• UNIQUE (A returned instance is a unique value.)
• ALL (All instances are returned. Duplicates are allowed.)

statement_set_operator :
• UNION (union)
• DIFFERENCE (difference)
• INTERSECTION | INTERSECT (intersection)
```

Example

```
CREATE TABLE nojoin_tbl_1 (ID INT, Name VARCHAR(32));

INSERT INTO nojoin_tbl_1 VALUES (1,'Kim');
INSERT INTO nojoin_tbl_1 VALUES (2,'Moy');
INSERT INTO nojoin_tbl_1 VALUES (3,'Jonas');
INSERT INTO nojoin_tbl_1 VALUES (4,'Smith');
INSERT INTO nojoin_tbl_1 VALUES (5,'Kim');
INSERT INTO nojoin_tbl_1 VALUES (6,'Smith');
INSERT INTO nojoin_tbl_1 VALUES (7,'Brown');

CREATE TABLE nojoin_tbl_2 (id INT, Name VARCHAR(32));
```

```
INSERT INTO nojoin_tbl_2 VALUES (5,'Kim');
INSERT INTO nojoin_tbl_2 VALUES (6,'Smith');
INSERT INTO nojoin_tbl_2 VALUES (7,'Brown');
INSERT INTO nojoin_tbl_2 VALUES (8,'Lin');
INSERT INTO nojoin_tbl_2 VALUES (9,'Edwin');
INSERT INTO nojoin_tbl_2 VALUES (10,'Edwin');

--Using UNION to get only distict rows
SELECT id, name FROM nojoin_tbl_1
UNION
SELECT id,name FROM nojoin_tbl_2;

           id  name
===================================
            1  'Kim'
            2  'Moy'
            3  'Jonas'
            4  'Smith'
            5  'Kim'
            6  'Smith'
            7  'Brown'
            8  'Lin'
            9  'Edwin'
           10  'Edwin'

--Using UNION ALL not eliminating duplicate selected rows
SELECT id, name FROM nojoin_tbl_1
UNION ALL
SELECT id,name FROM nojoin_tbl_2;

           id  name
===================================
            1  'Kim'
            2  'Moy'
            3  'Jonas'
            4  'Smith'
            5  'Kim'
            6  'Smith'
            7  'Brown'
            5  'Kim'
            6  'Smith'
            7  'Brown'
            8  'Lin'
            9  'Edwin'
           10  'Edwin'

--Using DEFFERENCE to get only rows returned by the first query but not by the second
SELECT id, name FROM nojoin_tbl_1
DIFFERENCE
SELECT id,name FROM nojoin_tbl_2;

           id  name
===================================
            1  'Kim'
            2  'Moy'
            3  'Jonas'
            4  'Smith'

--Using INTERSECTION to get only those rows returned by both queries
SELECT id, name FROM nojoin_tbl_1
INTERSECT
SELECT id,name FROM nojoin_tbl_2;

           id  name
===================================
            5  'Kim'
            6  'Smith'
            7  'Brown'
```

Containment Operators

Containment Operators

Description

Containment operators are used to check the containment relationship by performing comparison operation on operands of the set data type. Set data types or subqueries can be specified as operands. The operation returns TRUE or FALSE if there is a containment relationship between the two operands of identical/different/subset/proper subset.

The description and returned values about the containment operators supported by CUBRID are as follows:

Containment Operators Supported by CUBRID

Containment Operator	Description	Predicates	Return Value
A **SETEQ** B	A = B Elements in A and B are same each other.	{1,2} SETEQ {1,2,2}	0
A **SETNEQ** B	A ≠ B Elements in A and B are not same each other.	{1,2} SETNEQ {1,2,3}	1
A **SUPERSET** B	A ⊃ B B is a proper subset of A.	{1,2} SUPERSET {1,2,3}	0
A **SUBSET** B	A ⊂ B A is a proper subset of B.	{1,2} SUBSET {1,2,3}	1
A **SUPERSETEQ** B	A ⊇ B B is a subset of A.	{1,2} SUPERSETEQ {1,2,3}	0
A **SUBSETEQ** B	A ⊆ B A is a subset of B.	{1,2} SUBSETEQ {1,2,3}	1

The following table shows than possibility of operation by operand and type conversion if a containment operator is used.

Possibility of Operation by Operand

	SET	MULTISET	LIST(=SEQUENCE)
SET	Operation possible	Operation possible	Operation possible
MULTISET	Operation possible	Operation possible	Operation possible (**LIST** is converted into **MULTISET**)
LIST(=SEQUENCE)	Operation possible	Operation possible (**LIST** is converted into **MULTISET**)	Some operation possible (**SETEQ**, **SETNEQ**) Error occurs for the rest of operators.

Syntax

```
collection_operand  containment_operator  collection_operand

collection_operand:
• set
• multiset
• sequence(or list)
• subquery
• NULL

containment_operator:
• SETEQ
• SETNEQ
```

```
• SUPERSET
• SUBSET
• SUPERSETEQ
• SUBSETEQ
```

- *collection_operand* : This expression that can be specified as an operand is a single SET-valued attribute, an arithmetic expression containing a SET operator or a SET value enclosed in braces. If the type is not specified, the SET value enclosed in braces is treated as a **LIST** type by default.

  Subqueries can be specified as operands. If a column which is not a **SET** type is queried, a SET data type keyword is required for the subquery (e.g. **SET**(*subquery*)). The column retrieved by a subquery must return a single set so that it can be compared with the set of the other operands.

  If the element type is an object, the OIDs, not its contents, are compared. For example, two objects with different OIDs are considered to be different even though they have the same attribute values.

- **NULL** : Any of operands to be compared is **NULL**, **NULL** is returned.

```
--empty set is a subset of any set
EVALUATE ({} SUBSETEQ (CAST ({3,1,2} AS SET)));
       Result
=============
            1

--operation between set type and null returns null
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ NULL);
       Result
=============
        NULL

--{1,2,3} seteq {1,2,3} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SETEQ (CAST ({1,2,3,3} AS SET)));
       Result
=============
            1

--{1,2,3} seteq {1,2,3,3} returns false
EVALUATE ((CAST ({3,1,2} AS SET)) SETEQ (CAST ({1,2,3,3} AS MULTISET)));
       Result
=============
            0

--{1,2,3} setneq {1,2,3,3} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SETNEQ (CAST ({1,2,3,3} AS MULTISET)));
       Result
=============
            1

--{1,2,3} subseteq {1,2,3,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS SET)));
       Result
=============
            1

--{1,2,3} subseteq {1,2,3,4,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS MULTISET)));
       Result
=============
            1

--{1,2,3} subseteq {1,2,4,4,3} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS LIST)));
       Result
=============
            0

--{1,2,3} subseteq {1,2,3,4,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,3,4,4} AS LIST)));
       Result
=============
            1

--{3,1,2} seteq {3,1,2} returns true
EVALUATE ((CAST ({3,1,2} AS LIST)) SETEQ (CAST ({3,1,2} AS LIST)));
```

```
       Result
=============
            1
--error occurs because LIST subseteq LIST is not supported
EVALUATE ((CAST ({3,1,2} AS LIST)) SUBSETEQ (CAST ({3,1,2} AS LIST)));
       Result
=============
        error
```

SETEQ Operator

Description

The **SETEQ** operator returns **TRUE** if the first operand is the same as the second one. It can perform comparison operator for all collection data type.

Syntax

```
collection_operand SETEQ collection_operand
```

Example

```
--creating a table with SET type address column and LIST type zip_code column

CREATE TABLE contain_tbl (id int primary key, name char(10), address SET varchar(20),
zip_code LIST int);
INSERT INTO contain_tbl VALUES(1, 'Kim', {'country', 'state'},{1, 2, 3});
INSERT INTO contain_tbl VALUES(2, 'Moy', {'country', 'state'},{3, 2, 1});
INSERT INTO contain_tbl VALUES(3, 'Jones', {'country', 'state', 'city'},{1,2,3,4});
INSERT INTO contain_tbl VALUES(4, 'Smith', {'country', 'state', 'city',
'street'},{1,2,3,4});
INSERT INTO contain_tbl VALUES(5, 'Kim', {'country', 'state', 'city', 'street'},{1,2,3,4});
INSERT INTO contain_tbl VALUES(6, 'Smith', {'country', 'state', 'city',
'street'},{1,2,3,5});
INSERT INTO contain_tbl VALUES(7, 'Brown', {'country', 'state', 'city', 'street'},{});

--selecting rows when two collection_operands are same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SETEQ {'country','state',
'city'};
           id  name                  address               zip_code
===============================================================================
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}

1 row selected.

--selecting rows when two collection_operands are same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SETEQ {1,2,3};

           id  name                  address               zip_code
===============================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}

1 rows selected.
```

SETNEQ Operator

Description

The **SETNEQ** operator returns **TRUE(1)** if a first operand is different from a second operand. A comparable operation can be performed for all collection data types.

Syntax

```
collection_operand SETNEQ collection_operand
```

Example

```
--selecting rows when two collection_operands are not same in the WEHRE clause
```

```
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SETNEQ
{'country','state', 'city'};
           id  name                  address               zip_code
===============================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            2  'Moy       '          {'country', 'state'}  {3, 2, 1}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

6 rows selected.

--selecting rows when two collection_operands are not same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SETNEQ {1,2,3};
           id  name                  address               zip_code
===============================================================================
            2  'Moy       '          {'country', 'state'}  {3, 2, 1}
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}
```

SUPERSET Operator

Description

The **SUPERSET** operator returns **TRUE(1)** when a second operand is a proper subset of a first operand; that is, the first one is larger than the second one. If two operands are identical, **FALSE(0)** is returned. Note that **SUPERSET** is not supported if all operands are **LIST** type.

Syntax

```
collection_operand SUPERSET collection_operand
```

Example

```
--selecting rows when the first operand is a superset of the second operand and they are
not same
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUPERSET
{'country','state','city'};
           id  name                  address               zip_code
===============================================================================
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

--SUPERSET operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSET {1,2,3};

ERROR: ' superset ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUPERSET operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSET (CAST ({1,2,3}
AS SET));
           id  name                  address               zip_code
===============================================================================
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
```

SUPERSETEQ Operator

Description

The **SUPERSETEQ** operator returns **TRUE(1)** when a second operand is a subset of a first operand; that is, the first one is identical to or larger than the second one. Note that **SUPERSETEQ** is not supported if an operand is **LIST** type.

Syntax

```
collection_operand SUPERSETEQ collection_operand
```

Example

```
--selecting rows when the first operand is a superset of the second operand
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUPERSETEQ
{'country','state','city'};
           id  name                  address               zip_code
===============================================================================
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}


--SUPERSETEQ operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSETEQ {1,2,3};

ERROR: ' superseteq ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUPERSETEQ operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSETEQ (CAST
({1,2,3} AS SET));
           id  name                  address               zip_code
===============================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
```

SUBSET Operator

Description

The **SUBSET** operator returns **TRUE**(1) if the second operand contains all elements of the first operand. If the first and the second collection have the same elements, **FALSE**(0) is returned. Note that both operands are the **LIST** type, the **SUBSET** operation is not supported.

Syntax

```
collection_operand SUBSET collection_operand
```

Example

```
--selecting rows when the first operand is a subset of the second operand and they are not
same
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUBSET
{'country','state','city'};
           id  name                  address               zip_code
===============================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            2  'Moy       '          {'country', 'state'}  {3, 2, 1}

--SUBSET operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSET {1,2,3};

ERROR: ' subset ' operator is not defined on types sequence and sequence.
```

```
--Comparing operands with a SUBSET operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSET (CAST ({1,2,3}
AS SET));
           id  name                  address               zip_code
================================================================================
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}
```

SUBSETEQ Operator

Description

The **SUBSETEQ** operator returns **TRUE(1)** when a first operand is a subset of a second operand; that is, the second one is identical to or larger than the first one. Note that **SUBSETEQ** is not supported if an operand is **LIST** type.

Syntax

```
collection_operand SUBSETEQ collection_operand
```

Example

```
--selecting rows when the first operand is a subset of the second operand
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUBSETEQ
{'country','state','city'};
           id  name                  address               zip_code
================================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            2  'Moy       '          {'country', 'state'}  {3, 2, 1}
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}

3 rows selected.

--SUBSETEQ operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSETEQ {1,2,3};

ERROR: ' subseteq ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUBSETEQ operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSETEQ (CAST ({1,2,3}
AS SET));
           id  name                  address               zip_code
================================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

2 rows selected.
```

BIT Functions and Operators

Bitwise Operator

A **Bitwise** operator performs operations in bits, and can be used in arithmetic operations. An integer type is specified as the operand and the **BIT** type cannot be specified. An integer of **BIGINT** type (64-bit integer) is returned as the result of the operation. If one or more operand is **NULL**, **NULL** is returned.

The following table shows the bitwise operators supported by CUBRID.

The bitwise operators supported by CUBRID

Bitwise operator	Description	Expression	Return Value
&	Performs AND operation in bits and returns a BIGINT integer.	17 & 3	1
|	Performs OR operation in bits and returns a BIGINT integer.	17 | 3	19
^	Performs XOR operation in bits and returns a BIGINT integer.	17 ^ 3	18

~	A unary operator. It performs complementary operation that reverses (INVERT) the bit order of the operand and returns a BIGINT integer.	~17	-18
<<	Performs the operation of moving bits of the left operand as far to the left as the value of the right operand, and returns a BIGINT integer.	17 << 3	136
>>	Performs the operation of moving bits of the left operand as far to the right as the value of the right operand, and returns a BIGINT integer.	17 >> 3	2

BIT_AND Function

Description

An aggregate function. It performs **AND** operations in bits on every bit of *expr*. The return value is a **BIGINT** type. If there is no row that satisfies the expression, **NULL** is returned.

Syntax

```
BIT_AND(expr)
```

- *expr* : An expression of integer type

Example

```
CREATE TABLE bit_tbl(id int);
INSERT INTO bit_tbl VALUES (1), (2), (3), (4), (5);
SELECT 1&3&5, BIT_AND(id) FROM bit_tbl WHERE id in(1,3,5);
            1&3&5               bit_and(id)
============================================
                1                         1
```

BIT_OR Function

Description

An aggregate function. It performs **OR** operations in bits on every bit of *expr*. The return value is a **BIGINT** type. If there is no row that satisfies the expression, **NULL** is returned.

Syntax

```
BIT_OR(expr)
```

- *expr* : An expression of integer type

Example

```
SELECT 1|3|5, BIT_OR(id) FROM bit_tbl WHERE id in(1,3,5);
           1|3|5                   bit_or(id)
============================================
                     7                      7
```

BIT_XOR Function

Description

An aggregate function. It performs **XOR** operations in bits on every bit of *expr*. The return value is a **BIGINT** type. If there is no row that satisfies the expression, **NULL** is returned.

Syntax

```
BIT_XOR(expr)
```

- *expr* : An expression of integer type

Example

```
SELECT 1^2^3^, BIT_XOR(id) FROM bit_tbl WHERE id in(1,3,5);
    `        1^3^5               bit_xor(id)
============================================
                7                        7
```

BIT_COUNT Function

Description

The **BIT_COUNT** function returns the number of bits of *expr* that have been set to 1; it is not an aggregate function. The return value is a **BIGINT** type.

Syntax

```
BIT_COUNT (expr)
```

- *expr* : An expression of integer type

Example

```
SELECT BIT_COUNT(id) FROM bit_tbl WHERE id in(1,3,5);
   bit_count(id)
================
               1
               2
               2
```

String Functions and Operators

Concatenation Operator

Description

A concatenation operator gets a character string or bit string data type as an operand and returns a concatenated string. The plus sign (+) and double pipe symbol (||) are provided as concatenation operators for character string data. If **NULL** is specified as an operand, a **NULL** value is returned.

If **pipes_as_concat** that is a parameter related to SQL statement is set to **no**, a double pipe (||) symbol is interpreted as an **OR** operator. Therefore, to concatenate string or bit string, a plus operator (+) or the **CONCAT** function should be used.

Syntax

```
concat_operand1   +  concat_operand1
concat_operand2   ||  concat_operand2
concat_operand1 :
• bit string
• NULL

concat_operand2 :
• bit string
• character string
• NULL
```

- *concat_operand1* : Left string after concatenation. String or bit string can be specified.
- *concat_operand2* : Right string after concatenation. String or bit string can be specified.

Example

```
SELECT CONCAT('CUBRID', '2008' , 'R3.0');
   concat('CUBRID', '2008', 'R3.0')
======================
'CUBRID2008R3.0'
```

```
--it returns null when null is specified for one of parameters
SELECT CONCAT('CUBRID', '2008' , 'R3.0', NULL);
   concat('CUBRID', '2008', 'R3.0', null)
======================
  NULL


--it converts number types and then returns concatenated strings
SELECT CONCAT(2008, 3.0);
   concat(2008, 3.0)
======================
  '20083.0'
```

BIT_LENGTH Function

Description

The **BIT_LENGTH** function returns the length (bits) of a character string or bit string as an integer value. The return value of the **BIT_LENGTH** function may differ depending on the character set, because for the character string, the number of bytes taken up by a single character is different depending on the character set of the data input environment (e.g., EUC-KR: 2*8 bits). For details about character sets supported by CUBRID, see Definition and Characteristics.

Syntax

```
BIT_LENGTH ( string )

string :
• bit string
• character string
• NULL
```

- *string* : Specifies the character string or bit string whose number of bits is to be calculated. If this value is **NULL**, **NULL** is returned.

Example

```
SELECT BIT_LENGTH('');
   bit_length('')
=================
                0

SELECT BIT_LENGTH('CUBRID');
   bit_length('CUBRID')
=======================
                     48

SELECT BIT_LENGTH('큐브리드');
     bit_length('큐브리드')
=========================
                       64

SELECT BIT_LENGTH(B'010101010');
   bit_length(B'010101010')
===========================
                          9

CREATE TABLE bit_length_tbl (char_1 CHAR, char_2 CHAR(5), varchar_1 VARCHAR, bit_var_1 BIT
VARYING);
INSERT INTO bit_length_tbl VALUES('', '', '', B''); --Length of empty string
INSERT INTO bit_length_tbl VALUES('a', 'a', 'a', B'010101010'); --English character
INSERT INTO bit_length_tbl VALUES(NULL, '큐', '큐', B'010101010'); --Korean character and
NULL
INSERT INTO bit_length_tbl VALUES(' ', ' 큐', ' 큐', B'010101010'); --Korean character and
space

SELECT BIT_LENGTH(char_1), BIT_LENGTH(char_2), BIT_LENGTH(varchar_1), BIT_LENGTH(bit_var_1)
FROM bit_length_tbl;
```

```
bit_length(char_1)  bit_length(char_2)      bit_length(varchar_1)    bit_length(bit_var_1)
================================================================================
8                   40                         0                        0
8                   40                         8                        9
NULL                40                        16                        9
8                   40                        24                        9
```

CHAR_LENGTH, CHARACTER_LENGTH, LENGTHB, LENGTH Function

Description

CHAR_LENGTH, **LENGTHB**, and **LENGTH** are used interchangeably.

They return the length of a character string (byte) as an integer. The return value may be different depending on the character set (e.g., EUC-KR: 2 bites).

For details about the character sets supported by CUBRID, see Definition and Characteristics.

Syntax

```
CHAR_LENGTH( string )
CHARACTER_LENGTH( string )
LENGTHB( string )
LENGTH( string )

string :
• character string
• NULL
```

- *string* : Specifies the character string whose number of characters is to be calculated. If the character string is **NULL**, **NULL** is returned.

Remark

- The length of each space character that is included in a character string is one byte.
- For multi-byte strings, the length of a single character is calculated as 2 or 3 bytes depending on the character set of the data input environment.
- The length of empty quotes ('') to represent a space character is 0. Note that in a **CHAR**(*n*) type, the length of a space character is *n*, and it is specified as 1 if n is omitted.

Example

```
--character set is euc-kr for Korean characters
SELECT LENGTH('');
char length('')
==================
                 0

SELECT LENGTH('CUBRID');
char length('CUBRID')
==================
                 6

SELECT LENGTH('큐브리드');
char length('큐브리드')
==================
                 8

CREATE TABLE length_tbl (char_1 CHAR, char_2 CHAR(5), varchar_1 VARCHAR, varchar_2
VARCHAR);
INSERT INTO length_tbl VALUES('', '', '', ''); --Length of empty string
INSERT INTO length_tbl VALUES('a', 'a', 'a', 'a'); --English character
INSERT INTO length_tbl VALUES(NULL, '큐', '큐', '큐'); --Korean character and NULL
INSERT INTO length_tbl VALUES(' ', ' 큐', ' 큐', ' 큐'); --Korean character and space

SELECT LENGTH(char_1), LENGTH(char_2), LENGTH(varchar_1), LENGTH(varchar_2) FROM
length_tbl;
```

```
char_length(char_1) char_length(char_2) char_length(varchar_1) char_length(varchar_2)
================================================================================

1                     5                         0              0
1                     5                         1              1
NULL                  5                         2              2
1                     5                         3              3
```

CHR Function

Description

The **CHR** function returns a character that corresponds to the return value of the expression specified as an argument. It returns 0 if it exceeds range of character code.

Syntax

```
CHR( number_operand )
```

- *number_operand* : Specifies an expression that returns a numeric value.

Example

```
SELECT CHR(68) || CHR(68-2);
   chr(68)|| chr(68-2)
======================
  'DB'
```

CONCAT Function

Description

The **CONCAT** function has at least one argument specified for it and returns a string as a result of concatenating all argument values. The number of parameters that can be specified is unlimited. Automatic type casting takes place if a non-string type is specified as the argument. If any of the arguments is specified as **NULL**, **NULL** is returned.

If you want to insert separators between strings specified as arguments for concatenation, use the CONCAT_WS Function.

Syntax

```
CONCAT( string1, string2 [,string3 [, ... [, stringN]...]])

string :
• character string
• NULL
```

Example

```
SELECT CONCAT('CUBRID', '2008' , 'R3.0')

======================
'CUBRID2008R3.0'

--it returns null when null is specified for one of parameters
SELECT CONCAT('CUBRID', '2008' , 'R3.0', NULL)

======================
    NULL


--it converts number types and then returns concatenated strings
SELECT CONCAT(2008, 3.0)

======================
    '20083.0'
```

CONCAT_WS Function

Description

The **CONCAT_WS** function has at least two arguments specified for it. The function uses the first argument value as the separator and returns the result.

```
CONCAT_WS( string1, string2 [,string3 [, ... [, stringN]...]])

string :
• character string
• NULL
```

Example

```
SELECT CONCAT_WS(' ', 'CUBRID', '2008' , 'R3.0');
concat_ws(' ', 'CUBRID', '2008', 'R3.0')
======================
  'CUBRID 2008 R3.0'

--it returns strings even if null is specified for one of parameters
SELECT CONCAT_WS(' ', 'CUBRID', '2008', NULL, 'R3.0');
concat_ws(' ', 'CUBRID', '2008', null, 'R3.0')
======================
  'CUBRID 2008 R3.0'

--it converts number types and then returns concatenated strings with separator
SELECT CONCAT_WS(' ',2008, 3.0);
concat_ws(' ', 2008, 3.0)
======================
  '2008 3.0'
```

ELT Function

Description

If *N* is 1, the **ELT** function returns *string1* and if *N* is 2, it returns *string2*. The return value is a **VARCHAR** type. You can add conditional expressions as needed.

The maximum length of the character string is 33,554,432 and if this length is exceeded, **NULL** will be returned.

If *N* is 0 or a negative number, an empty string will be returned. If *N* is greater than the number of this input character string, **NULL** will be returned as it is out of range. If *N* is a type that can not be converted to an integer, an error will be returned.

Syntax

```
ELT(N, string1, string2, ... )

string :
• character string
• NULL
```

Example

```
SELECT ELT(3,'string1','string2','string3');
  elt(3, 'string1', 'string2', 'string3')
======================
  'string3'

SELECT ELT('3','1/1/1','23:00:00','2001-03-04');
  elt('3', '1/1/1', '23:00:00', '2001-03-04')
======================
  '2001-03-04'

SELECT ELT(-1, 'string1','string2','string3');
  elt(-1, 'string1','string2','string3')
======================
  NULL
```

```
SELECT ELT(4,'string1','string2','string3');
  elt(4, 'string1', 'string2', 'string3')
======================
  NULL

SELECT ELT(3.2,'string1','string2','string3');
  elt(3.2, 'string1', 'string2', 'string3')
======================
  'string3'

SELECT ELT('a','string1','string2','string3');

ERROR: Cannot coerce value of domain "character" to domain "bigint".
```

FIELD Function

Description

The **FIELD** function returns the location index value (position) of a string of *string1*, *string2*. The function returns 0 if it does not have a parameter value which is the same as *search_string*. It returns 0 if *search_string* is **NULL** because it cannot perform the comparison operation with the other arguments.

If all arguments specified for **FIELD()** are of string type, string comparison operation is performed: if all of them are of number type, numeric comparison operation is performed. If the type of one argument is different from that of another, a comparison operation is performed by casting each argument to the type of the first argument. If type casting fails during the comparison operation with each argument, the function considers the result of the comparison operation as **FALSE** and resumes the other operations.

Syntax

```
FIELD( search_string, string1 [,string2 [, ... [, stringN]...]])

string :
• character string
• NULL
```

Example

```
SELECT FIELD('abc', 'a', 'ab', 'abc', 'abcd', 'abcde');
   field('abc', 'a', 'ab', 'abc', 'abcd', 'abcde')
==================================================
                                                 3

--it returns 0 when no same string is found in the list
SELECT FIELD('abc', 'a', 'ab', NULL);
   field('abc', 'a', 'ab', null)
================================
                               0

--it returns 0 when null is specified in the first parameter
SELECT FIELD(NULL, 'a', 'ab', NULL);
   field(null, 'a', 'ab', null)
===============================
                              0

SELECT FIELD('123', 1, 12, 123.0, 1234, 12345);
   field('123', 1, 12, 123.0, 1234, 12345)
==========================================
                                         0

SELECT FIELD(123, 1, 12, '123.0', 1234, 12345);
   field(123, 1, 12, '123.0', 1234, 12345)
==============================================
                                             3
```

INSERT Function

Description

The **INSERT** function inserts a partial character string as long as the length from the specific location of the input character string. The return value is a **VARCHAR** type.

The maximum length of the character string is 33,554,432 and if this length is exceeded, **NULL** will be returned.

Syntax

```
INSERT( str, pos, len, string )
```

- *str* : Input character string
- *pos* : *str* location. Starts from 1. If *pos* is smaller than 1 or greater than the length of *string* + 1, the *string* will not be inserted and the *str* will be returned instead.
- *len* : Length of *string* to insert *pos* of *str*. If *len* exceeds the length of the partial character string, insert as many values as *string* in the *pos* of the *str*. If *len* is a negative number, *str* will be the end of the character string.
- *string* : Partial character string to insert to *str*

Example

```
SELECT INSERT('cubrid',2,2,'dbsql');
  insert('cubrid', 2, 2, 'dbsql')
======================
  'cdbsqlrid'

SELECT INSERT('cubrid',0,3,'db');
  insert('cubrid', 0, 3, 'db')
======================
  'cubrid'

SELECT INSERT('cubrid',-3,3,'db');
  insert('cubrid', -3, 3, 'db')
======================
  'cubrid'

SELECT INSERT('cubrid',3,100,'db');
  insert('cubrid', 3, 100, 'db')
======================
  'cudb'

SELECT INSERT('cubrid',7,100,'db');
  insert('cubrid', 7, 100, 'db')
======================
  'cubriddb'

SELECT INSERT('cubrid',3,-1,'db');
  insert('cubrid', 3, -1, 'db')
======================
  'cudb'
```

INSTR Function

Description

The **INSTR** function, similarly to the **POSITION**, returns the position of a *substring* within *string*; the position. For the **INSTR** function, you can specify the starting position of the search for *substring* to make it possible to search for duplicate *substring*.

Note that the function calculates the starting position and the length of the character string in bytes, not in characters. For a multi-byte character set, the number of bite representing onc character is different, so the return value may not be the same.

Syntax

```
INSTR( string , substring [, position] )
```

```
string , substring :
• character string
• NULL
position :
• INT
• NULL
```

- *string* : Specifies the input character string.
- *substring* : Specifies the character string whose position is to be returned.
- *position* : Optional. Represents the position of a *string* where the search begins. If omitted, the default value 1 is applied. The first position of the *string* is specified as 1. If the value is negative, the system counts backward from the end of the *string*.

Example

```
--character set is euc-kr for Korean characters
--it returns position of the first 'b'
SELECT INSTR ('12345abcdeabcde','b');
   instr('12345abcdeabcde', 'b', 1)
===================================
                                  7

-- it returns position of the first '나' on double byte charset
SELECT INSTR ('12345가나다라마가나다라마', '나' );
   instr('12345가나다라마가나다라마', '나', 1)
=================================
                                8

-- it returns position of the second '나' on double byte charset
SELECT INSTR ('12345가나다라마가나다라마', '나', 16 );
   instr('12345가나다라마가나다라마', '나', 16)
=================================
                               18

--it returns position of the 'b' searching from the 8th position
SELECT INSTR ('12345abcdeabcde','b', 8);
   instr('12345abcdeabcde', 'b', 8)
===================================
                                 12

--it returns position of the 'b' searching backwardly from the end
SELECT INSTR ('12345abcdeabcde','b', -1);
   instr('12345abcdeabcde', 'b', -1)
====================================
                                  12

--it returns position of the 'b' searching backwardly from a specified position
SELECT INSTR ('12345abcdeabcde','b', -8);
   instr('12345abcdeabcde', 'b', -8)
====================================
                                   7
```

LCASE, LOWER Function

Description

The **LOWER** function converts uppercase characters that are included in a character string to lowercase characters; it works the same as the **LCASE** function. Note that the **LOWER** function may not work properly in character sets that are not supported by CUBRID. For details about the character sets supported by CUBRID, see Definition and Characteristics.

Syntax

```
LCASE (string )
LOWER ( string )
```

```
string :
• character string
• NULL
```

- *string* : Specifies the string in which uppercase characters are to be converted to lowercase. If the value is **NULL**, **NULL** is returned.

Example

```
SELECT LOWER('');
  lower('')
======================
  ''

SELECT LOWER(NULL);
  lower(null)
======================
  NULL

SELECT LOWER('Cubrid');
  lower('Cubrid')
======================
  'cubrid'
```

LEFT Function

Description

The **LEFT** function returns a length number of characters from the leftmost of *string*. If any of the arguments is **NULL**, **NULL** is returned. If a value greater than the *length* of the *string* or a negative number is specified for a length, the entire string is returned.

To extract a length number of characters from the rightmost of the string, use the RIGHT Function.

Syntax

```
LEFT( string , length )

string :
• character string
• NULL

length :
• INT
• NULL
```

Example

```
SELECT LEFT('CUBRID', 3);
 left('CUBRID', 3)
======================
  'CUB'

SELECT LEFT('CUBRID', 10);
  left('CUBRID', 10)
======================
  'CUBRID'
```

LOCATE Function

Description

The **LOCATE** function returns the location index value of a `substring` within a character string. The third argument *position* can be omitted. If this argument is specified, the function searches for **substring** from the given position and returns the location index value of the first occurrence. If the *substring* cannot be found within the string, 0 is returned.

The **LOCATE** function is working like the POSITION Function, but you cannot use **LOCATE** for bit strings.

Syntax

```
LOCATE ( substring, string [, position] )

string :
• character string
• NULL
```

Example

```
--it returns 1 when substring is empty space
SELECT LOCATE ('', '12345abcdeabcde');
===============================
                              1

--it returns position of the first 'abc'
SELECT LOCATE ('abc', '12345abcdeabcde');
================================
                               6

--it returns position of the second 'abc'
SELECT LOCATE ('abc', '12345abcdeabcde', 8) FROM db_root;
======================================
                                    11

--it returns 0 when no substring found in the string
SELECT LOCATE ('ABC', '12345abcdeabcde');
=================================
                                0
```

LPAD Function

Description

The **LPAD** function pads the left side of a string with a specific set of characters.

Syntax

```
LPAD( char1, n, [, char2 ] )

char1 :
• character string
• string valued column
• NULL

n :
• integer
• NULL

char2 :
• character string
• NULL
```

- *char1* : Specifies the string to pad characters to. If *n* is smaller than the length of *char1*, padding is not performed, and *char1* is truncated to length n and then returned. A single character is processed as 2 or 3 bytes in multi-byte character set environment. If *char1* is truncated up to the first byte representing a character according to a value of *n*, the last byte is removed and a space character (1 byte) is added to the left because the last character cannot be represented normally. If the value is **NULL**, **NULL** is returned.
- *n* : Specifies the total length of *char1* in bytes. Note that the number and the length of the character strings may be different in multi-byte character set environment. If the value is **NULL**, **NULL** is returned.
- *char2* : Specifies the string to pad to the left until the length of *char1* reaches *n*. If it is not specified, empty characters (' ') are used as a default. If the value is **NULL**, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters

--it returns only 3 characters if not enough length is specified
SELECT LPAD ('CUBRID', 3, '?');
  lpad('CUBRID', 3, '?')
```

```
======================
  'CUB'

--on multi-byte charset, it returns the first character only with a left padded space
SELECT LPAD ('큐브리드', 3, '?');
 lpad('큐브리드', 3, '?')
======================
  ' 큐'

--padding spaces on the left till char_length is 10
SELECT LPAD ('CUBRID', 10);
 lpad('CUBRID', 10)
======================
  '    CUBRID'

--padding specific characters on the left till char_length is 10
SELECT LPAD ('CUBRID', 10, '?');
 lpad('CUBRID', 10, '?')
======================
  '????CUBRID'

--padding specific characters on the left till char_length is 10
SELECT LPAD ('큐브리드', 10, '?');
 lpad('큐브리드', 10, '?')
======================
  '??큐브리드'

--padding 4 characters on the left
SELECT LPAD ('큐브리드', LENGTH('큐브리드')+4, '?');
 lpad('큐브리드',  char_length('큐브리드')+4, '?')
======================
  '????큐브리드'
```

LTRIM Function

Description

The **LTRIM** function removes all specified characters from the left-hand side of a string.

Syntax

```
LTRIM( string [, trim_string])

string :
• character string
• string valued column
• NULL

trim_string :
• character string
• NULL
```

- *string* : Enters a string or string-type column to trim. If this value is **NULL**, **NULL** is returned.
- *trim_string* : You can specify a specific string to be removed in the left side of *string*. If it is not specified, empty characters (' ') is automatically specified so that the empty characters in the left side are removed.

Example

```
--trimming spaces on the left
SELECT LTRIM ('     Olympic     ');
  ltrim('     Olympic     ')
======================
  'Olympic     '

--If NULL is specified, it returns NULL
SELECT LTRIM ('iiiiiOlympiciiiii', NULL);
```

```
  ltrim('iiiiiOlympiciiiii', null)
======================
  NULL

-- trimming specific strings on the left
SELECT LTRIM ('iiiiiOlympiciiiii', 'i');
  ltrim('iiiiiOlympiciiiii', 'i')
======================
  'Olympiciiiii'
```

MID Function

Description

The **MID** function extracts a string with the length of *substring_length* from a *position* within the *string* and then returns it. If a negative number is specified as a *position* value, the *position* is calculated in a reverse direction from the end of the *string*. **substring_length** cannot be omitted. If a negative value is specified, the function considers this as 0 and returns an empty string.

The **MID** function is working like the SUBSTR Function, but there are differences in that it cannot be used for bit strings, that the *substring_length* argument must be specified, and that it returns an empty string if a negative number is specified for *substring_length*.

Syntax

```
string :
• character string
• NULL

position :
• integer
• NULL

substring_length :
• integer
• NULL
```

- *string* : Specifies an input character string. If this value is **NULL**, **NULL** is returned.
- *position* : Specifies the starting position from which the string is to be extracted. The position of the first character is 1. It is considered to be 1 even if it is specified as 0. If the input value is **NULL**, **NULL** is returned.
- *substring_lenghth* : Specifies the length of the string to be extracted. If 0 or a negative number is specified, an empty string is returned; if **NULL** is specified, **NULL** is returned.

Example

```
CREATE TABLE mid_tbl(a VARCHAR);
INSERT INTO mid_tbl VALUES('12345abcdeabcde');

--it returns empty string when substring_length is 0
SELECT MID(a, 6, 0), SUBSTR(a, 6, 0), SUBSTRING(a, 6, 0) FROM mid_tbl;
  mid(a, 6, 0)          substr(a, 6, 0)       substring(a from 6 for 0)
==================================================================
  ''                    ''                    ''

--it returns 4-length substrings counting from the 6th position
SELECT MID(a, 6, 4), SUBSTR(a, 6, 4), SUBSTRING(a, 6, 4) FROM mid_tbl;
  mid(a, 6, 4)          substr(a, 6, 4)       substring(a from 6 for 4)
==================================================================
  'abcd'                'abcd'                'abcd'

--it returns a empty string when substring_length < 0
SELECT MID(a, 6, -4), SUBSTR(a, 6, -4), SUBSTRING(a, 6, -4) FROM mid_tbl;
  mid(a, 6, -4)         substr(a, 6, -4)      substring(a from 6 for -4)
==================================================================
  ''                    NULL                  'abcdeabcde'

--it returns 4-length substrings at 6th position counting backward from the end
SELECT MID(a, -6, 4), SUBSTR(a, -6, 4), SUBSTRING(a, -6, 4) FROM mid_tbl;
```

```
  mid(a, -6, 4)         substr(a, -6, 4)      substring(a from -6 for 4)
==================================================================
  'eabc'                'eabc'                '1234'
```

OCTET_LENGTH Function

Description

The **OCTET_LENGTH** function returns the length (byte) of a character string or bit string as an integer. Therefore, it returns 1 (byte) if the length of the bit string is 8 bits, but 2 (bytes) if the length is 9 bits.

Syntax

```
OCTET_LENGTH ( string )

string :
• bit string
• character string
• NULL
```

- *string* : Specifies the character or bit string whose length is to be returned in bytes. If the value is **NULL**, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters

SELECT OCTET_LENGTH('');
 octet_length('')
==================
                 0

SELECT OCTET_LENGTH('CUBRID');
 octet_length('CUBRID')
==================
                 6

SELECT OCTET_LENGTH('큐브리드');
 octet_length('큐브리드')
==================
                 8

SELECT OCTET_LENGTH(B'010101010');
 octet_length(B'010101010')
==================
                 2

CREATE TABLE octet_length_tbl (char_1 CHAR, char_2 CHAR(5), varchar_1 VARCHAR, bit_var_1
BIT VARYING);
INSERT INTO octet_length_tbl VALUES('', '', '', B''); --Length of empty string
INSERT INTO octet_length_tbl VALUES('a', 'a', 'a', B'010101010'); --English character
INSERT INTO octet_length_tbl VALUES(NULL, '큐', '큐', B'010101010'); --Korean character and
NULL
INSERT INTO octet_length_tbl VALUES(' ', ' 큐', ' 큐', B'010101010'); --Korean character
and space

SELECT OCTET_LENGTH(char_1), OCTET_LENGTH(char_2), OCTET_LENGTH(varchar_1),
OCTET_LENGTH(bit_var_1) FROM octet_length_tbl;
octet_length(char_1) octet_length(char_2) octet_length(varchar_1) octet_length(bit_var_1)
================================================================================
1                      5                         0                        0
1                      5                         1                        2
NULL                   5                         2                        2
1                      5                         3                        2
```

POSITION Function

Description

The **POSITION** function returns the position of a character string corresponding to *substring* within a character string corresponding to *string*. Note that it returns the position in bytes, not in characters. Therefore, the return values may differ because the number of bytes representing a single character is different in multi-byte character sets.

An expression that returns a character string or a bit string can be specified as an argument of this function. The return value is an integer greater than or equal to 0. This function returns the position value in bytes for a character string, and in bits for a bit string.

The **POSITION** function is occasionally used in combination with other functions. For example, if you want to extract a certain string from another string, you can use the result of the **POSITION** function as an input to the **SUBSTRING** function.

Syntax

```
POSITION ( substring IN string )

substring :
• bit string
• character string
• NULL
```

- *substring* : Specifies the character string whose position is to be returned. If the value is an empty character, 1 is returned. If the value is **NULL**, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters

--it returns 1 when substring is empty space
SELECT POSITION ('' IN '12345abcdeabcde');
  position('' in '12345abcdeabcde')
===============================
                              1

--it returns position of the first 'b'
SELECT POSITION ('b' IN '12345abcdeabcde');
  position('b' in '12345abcdeabcde')
================================
                               7

-- it returns position of the first '나' on double byte charset
SELECT POSITION ('나' IN '12345가나다라마가나다라마');
  position('나' in '12345가나다라마가나다라마')
=================================
                                8

--it returns 0 when no substring found in the string
SELECT POSITION ('f' IN '12345abcdeabcde');
  position('f' in '12345abcdeabcde')
=================================
                                0

SELECT POSITION (B'1' IN B'000011110000');
  position(B'1' in B'000011110000')
=================================
                                5
```

REPEAT Function

Description

The **REPEAT** function returns the character string with a length equal to the number of repeated input character strings. The return value is a **VARCHAR** type. The maximum length of the characcter string is 33,554,432 and if it this length is exceeded, **NULL** will be returned. If one of the parameters is **NULL**, **NULL** will be returned.

Syntax

```
REPEAT( string, count )
```

- *string* : Character string
- *count* : Repeat count. If you enter 0 or a negative number, an empty string will be returned and if you enter a non-numeric data type, an error will be returned.

Example

```
SELECT REPEAT('cubrid',3);
   repeat('cubrid', 3)
======================
  'cubridcubridcubrid'

SELECT REPEAT('cubrid',32000000);
   repeat('cubrid', 32000000)
======================
  NULL

SELECT REPEAT('cubrid',-1);
   repeat('cubrid', -1)
======================
  ''

SELECT REPEAT('cubrid','a');
ERROR: Cannot coerce value of domain "character" to domain "integer".
```

REPLACE Function

Description

The **REPLACE** function searches for a character string, *search_string*, within a given character string, *string*, and replaces it with a character string, *replacement_string*. If the string to be replaced, *replacement_string* is omitted, all *search_strings* retrieved from *string* are removed. If **NULL** is specified as an argument, **NULL** is returned.

Syntax

```
REPLACE( string, search_string [, replacement_string ] )

string :
• character string
• NULL

search_string :
• character string
• NULL

replacement_string :
• character string
• NULL
```

- *string* : Specifies the original string. If the value is **NULL**, **NULL** is returned.
- *search_string* : Specifies the string to be searched. If the value is **NULL**, **NULL** is returned.
- *replacement_string* : Specifies the string to replace the *search_string*. If this value is omitted, *string* is returned with the *search_string* removed. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns NULL when an argument is specified with NULL value
SELECT REPLACE('12345abcdeabcde','abcde',NULL);

======================
  NULL

--not only the first substring but all substrings into 'ABCDE' are replaced
SELECT REPLACE('12345abcdeabcde','abcde','ABCDE');

======================
  '12345ABCDEABCDE'

--it removes all of substrings when replace_string is omitted
SELECT REPLACE('12345abcdeabcde','abcde');

======================
  '12345'
```

REVERSE Function

Description

The **REVERSE** function returns *string* converted in the reverse order.

Syntax

```
REVERSE( string )

string :
• character string
• NULL
```

- *substring* : Specifies an input character string. If the value is an empty string, empty value is returned. If the value is **NULL**, **NULL** is returned.

Example

```
SELECT REVERSE('CUBRID');
  reverse ('CUBRID')
======================
  'DIRBUC'
```

RIGHT Function

Description

The **RIGHT** function returns a *length* number of characters from the rightmost of a *string*. If any of the arguments is **NULL**, **NULL** is returned. If a value greater than the length of the *string* or a negative number is specified for a *length*, the entire string is returned.

To extract a length number of characters from the leftmost of the string, use the LEFT Function.

Syntax

```
RIGHT( string , length )

string :
• character string
• NULL

length :
• INT
• NULL
```

Example

```
SELECT RIGHT('CUBRID', 3);
```

```
 right('CUBRID', 3)
======================
  'RID'

SELECT RIGHT ('CUBRID', 10);
 right('CUBRID', 10)
======================
  'CUBRID'
```

RPAD Function

Description

The **RPAD** function pads the right side of a string with a specific set of characters.

Syntax

```
RPAD( char1, n, [, char2 ] )

char1 :
• character string
• string valued column
• NULL

n :
• integer
• NULL

char2 :
• character string
• NULL
```

- *char1* : Specifies the string to pad characters to. If n is smaller than the length of *char1*, padding is not performed, and *char1* is truncated to length n and then returned. A single character is processed as 2 or 3 bytes in multi-byte character set environment. If *char1* is truncated up to the first byte representing a character according to a value of *n*, the last byte is removed and an empty character (1 byte) is added to the left because the last character cannot be represented normally. If the value is **NULL**, **NULL** is specified.
- *n* : Specifies the total length of *char1* in bytes. Note that the number and the length of the character strings may be different in multi-byte character set environment. If the value is **NULL**, **NULL** is specified.
- *char2* : Specifies the string to pad to the right until the length of *char1* reaches *n*. If it is not specified, empty characters (' ') are used as a default. If the value is **NULL**, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters

--it returns only 3 characters if not enough length is specified

SELECT RPAD ('CUBRID', 3, '?');

======================
  'CUB'

--on multi-byte charset, it returns the first character only with a right-padded space

SELECT RPAD ('큐브리드', 3, '?');
======================
  '큐 '

--padding spaces on the right till char_length is 10
SELECT RPAD ('CUBRID', 10);

======================
  'CUBRID    '

--padding specific characters on the right till char_length is 10

SELECT RPAD ('CUBRID', 10, '?');
======================
```

```
  'CUBRID????'

--padding specific characters on the right till char_length is 10

SELECT RPAD ('큐브리드', 10, '?');
======================
  '큐브리드??'

--padding 4 characters on the right

SELECT RPAD ('큐브리드', LENGTH('큐브리드')+4, '?');
======================
  '큐브리드????'
```

RTRIM Function

Description

The **RTRIM** function removes specified characters from the right-hand side of a string.

Syntax

```
RTRIM( string [, trim_string])

string :
• character string
• string valued column
• NULL

trim_string :
• character string
• NULL
```

- *string* : Enters a string or string-type column to trim. If this value is **NULL**, **NULL** is returned.
- *trim_string* : You can specify a specific string to be removed in the right side of *string*. If it is not specified, empty characters (' ') is automatically specified so that the empty characters in the right side are removed.

Example

```
SELECT RTRIM ('     Olympic     ');
 rtrim('     Olympic     ')
======================
  '     Olympic'

--If NULL is specified, it returns NULL
SELECT RTRIM ('iiiiiOlympiciiiii', NULL);
 rtrim('iiiiiOlympiciiiii', null)
======================
  NULL

-- trimming specific strings on the right
SELECT RTRIM ('iiiiiOlympiciiiii', 'i');
 rtrim('iiiiiOlympiciiiii', 'i')
======================
  'iiiiiOlympic'
```

SPACE Function

Description

The **SPACE** function returns as many empy strings as the number specified. The return value is a **VARCHAR** type.

Syntax

```
SPACE(N)
```

- *N* : Space count. Can not be greater than the value specified in the system parameter, **string_max_size_bytes** (default 1048576). If it exceeds the specified value, **NULL** will be returned. The maximum value is 33,554,432; if this length is exceeded, **NULL** will be returned. If you enter 0 or a negative number, an empty string will be returned; if you enter a type that can't be converted to a numeric value, an error will be returned.

Example

```
SELECT SPACE(8);
   space(8)
======================
  '        '

SELECT LENGTH(space(1048576));
   char_length( space(1048576))
===============================
                        1048576

SELECT LENGTH(space(1048577));
   char_length( space(1048577))
===============================
                           NULL

-- string_max_size_bytes=33554432
SELECT LENGTH(space('33554432'));
   char_length( space('33554432'))
==================================
                          33554432

SELECT SPACE('aaa');

ERROR: Cannot coerce value of domain "character" to domain "bigint".
```

STRCMP Function

Description

The **STRCMP** function compares two strings, *string1* and *string2*, and returns 0 if they are identical, 1 if *string1* is greater, or -1 if *string1* is smaller. If any of the parameters is **NULL**, **NULL** is returned.

Syntax

```
STRCMP( string1 , string2 )

string :
• character string
• NULL
```

Example

```
SELECT STRCMP('abc', 'abc');

=======================
                      0
SELECT STRCMP ('acc', 'abc');

=======================
                      1

--STRCMP works case-insensitively
SELECT STRCMP ('ABC','abc');

=======================
                      0
```

SUBSTR Function

Description

The **SUBSTR** function extracts a character string with the length of *substring_length* from a position, *position*, within character string, *string*, and then returns it. If a negative number is specified as a *position* value, the position is calculated in a reverse direction from the end of the string. If *substring_length* is omitted, character strings between the given position, *position*, and the end of the string are extracted, and then returned.

Note that it returns the starting position and the length of character string in bytes, not in characters. Therefore, in a multi-byte character set, you must specify the parameter in consideration of the number of bytes representing a single character.

Syntax

```
SUBSTR( string, position [, substring_length])

string :
• character string
• bit string
• NULL

position :
• integer
• NULL

substring_length :
• integer
```

- *string* : Specifies the input character string. If the input value is **NULL**, **NULL** is returned.
- *position* : Specifies the position from where the string is to be extracted in bytes. Even though the position of the first character is specified as 1 or a negative number, it is considered as 1. If a value greater than the string length or **NULL** is specified, **NULL** is returned.
- *substring_length* : Specifies the length of the string to be extracted in bytes. If this argument is omitted, character strings between the given position, *position*, and the end of them are extracted. **NULL** cannot be specified as an argument value of this function. If 0 is specified, an empty string is returned; if a negative value is specified, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters

--it returns empty string when substring_length is 0
SELECT SUBSTR('12345abcdeabcde',6, 0);
 substr('12345abcdeabcde', 6, 0)
======================
  ''

--it returns 4-length substrings counting from the position
SELECT SUBSTR('12345abcdeabcde', 6, 4), SUBSTR('12345abcdeabcde', -6, 4);
 substr('12345abcdeabcde', 6, 4)   substr('12345abcdeabcde', -6, 4)
============================================
  'abcd'                'eabc'

--it returns substrings counting from the position to the end
SELECT SUBSTR('12345abcdeabcde', 6), SUBSTR('12345abcdeabcde', -6);
 substr('12345abcdeabcde', 6)   substr('12345abcdeabcde', -6)
============================================
  'abcdeabcde'          'eabcde'

-- it returns 4-length substrings counting from 16th position on double byte charset
SELECT SUBSTR ('12345가나다라마가나다라마', 16 , 4);
 substr('12345가나다라마가나다라마', 16 , 4)
======================
  '가나'
```

SUBSTRING Function

Description

The **SUBSTRING** function, operating like **SUBSTR**, extracts a character string having the length of *substring_length* from a position, *position*, within character string, *string*, and returns it.

If a negative number is specified to the *position* value, the **SUBSTRING** function calculates the position from the beginning of the string. And **SUBSTR** function calculates the position from the end of the string. If a negative number is specified to the *substring_length* value, the **SUBSTRING** function handles the argument is omitted, but the **SUBSTR** function returns **NULL**.

Syntax

```
SUBSTRING( string, position [, substring_length])
SUBSTRING( string FROM position [FOR substring_length] )

string :
• bit string
• character string
• NULL

position :
• integer
• NULL

substring_length :
• integer
```

- *string* : Specifies the input character string. If the input value is **NULL**, **NULL** is returned.
- *position* : Specifies the position from where the string is to be extracted in bytes. Even though the position of the first character is specified as 1 or a negative number, it is considered as 1. If a value greater than the string length is specified, an empty string is returned. If **NULL**, **NULL** is returned.
- *substring_length* : Specifies the length of the string to be extracted in bytes. If this argument is omitted, character strings between the given position, *position*, and the end of them are extracted. **NULL** cannot be specified as an argument value of this function. If 0 is specified, an empty string is returned; if a negative value is specified, **NULL** is returned.

Example

```
SELECT SUBSTRING('12345abcdeabcde', -6 ,4), SUBSTR('12345abcdeabcde', -6 ,4);
============================================
  '1234'                'eabc'


SELECT SUBSTRING('12345abcdeabcde', 16), SUBSTR('12345abcdeabcde', 16);
============================================
  ''                    NULL

SELECT SUBSTRING('12345abcdeabcde', 6, -4), SUBSTR('12345abcdeabcde', 6, -4);
============================================
  'abcdeabcde'          NULL
```

SUBSTRING_INDEX Function

Description

The **SUBSTRING_INDEX** function counts the separators included in the partial character string and will return the partial character string before *count*th. The return value is a **VARCHAR** type.

Syntax

```
SUBSTRING_INDEX (string, delim, count)
```

- *string* : Input character string. The maximum length is 33,554,432 and if this length is exceeded, **NULL** will be returned.

- *delim* : Delimiter. It is case-sensitive.
- *count* : Delimiter occurrence count. If you enter a positive number, it counts the character string from the left and if you enter a negative number, it counts it from the right. If it is 0, an empty string will be returned. If the type can not be converted, an error wll be returned.

Example

```
SELECT SUBSTRING_INDEX('www.cubrid.org','.','2');
  substring_index('www.cubrid.org', '.', '2')
======================
  'www.cubrid'

SELECT SUBSTRING_INDEX('www.cubrid.org','.','2.3');
  substring_index('www.cubrid.org', '.', '2.3')
======================
  'www.cubrid'

SELECT SUBSTRING_INDEX('www.cubrid.org',':','2.3');
  substring_index('www.cubrid.org', ':', '2.3')
======================
  'www.cubrid.org'

SELECT SUBSTRING_INDEX('www.cubrid.org','cubrid',1);
  substring_index('www.cubrid.org', 'cubrid', 1)
======================
  'www.'

SELECT SUBSTRING_INDEX('www.cubrid.org','.',100);
  substring_index('www.cubrid.org', '.', 100)
======================
  'www.cubrid.org'
```

TRANSLATE Function

Description

The **TRANSLATE** function searches for a character specified as a character string, *from_substring*, within a given character string, *string*, and replaces it with a character specified as a character string, *to_substring*, if exists. Correspondence relationship is determined according to the order of characters specified by *from_substring* and *to_substring*. All characters in *from_substring* that do not have a one to one correspondence relationship with the characters in *to_substring* are removed from the string. The **TRANSLATE** function is working like the **REPLACE** function, but you cannot omit the to_substring argument with this function.

Syntax

```
TRANSLATE( string, from_substring, to_substring )

string :
• character string
• NULL

from_substring :
• character string
• NULL

to_substring :
• character string
• NULL
```

- *string* : Specifies the original string. If the value is **NULL**, **NULL** is returned.
- *from_substring* : Specifies the string to be retrieved. If the value is **NULL**, **NULL** is returned.
- *to_substring* : Specifies the character string in the *from_substring* to be replaced. It cannot be omitted. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns NULL when an argument is specified with NULL value
```

```
SELECT TRANSLATE('12345abcdeabcde','abcde', NULL);
  translate('12345abcdeabcde', 'abcde', null)
======================
  NULL

--it translates 'a','b','c','d','e' into '1', '2', '3', '4', '5' respectively
SELECT TRANSLATE('12345abcdeabcde', 'abcde', '12345');
  translate('12345abcdeabcde', 'abcde', '12345')
======================
  '123451234512345'

--it translates 'a','b','c' into '1', '2', '3' respectively and removes 'd's and 'e's
SELECT TRANSLATE('12345abcdeabcde','abcde', '123');
  translate('12345abcdeabcde', 'abcde', '123')
======================
  '12345123123'

--it removes 'a's,'b's,'c's,'d's, and 'e's in the string
SELECT TRANSLATE('12345abcdeabcde','abcde', '');
  translate('12345abcdeabcde', 'abcde', '')
======================
  '12345'

--it only translates 'a','b','c' into '3', '4', '5' respectively
SELECT TRANSLATE('12345abcdeabcde','ABabc', '12345');
  translate('12345abcdeabcde', 'ABabc', '12345')
======================
  '12345345de345de'
```

TRIM Function

Description

The **TRIM** function trims leading and/or trailing characters from a character string.

Syntax

```
TRIM ( [ [ LEADING | TRAILING | BOTH ] [ trim_string ] FROM ] string )

trim_string :
• character string
• NULL

string :
• character string literal
• string valued column
• NULL
```

- *trim_string* : You can specify a specific string to be removed that is in front of or at the back of the target string. If it is not specified, an empty character (' ') is automatically specified so that spaces in front of or at the back of the target string are removed.
- *string* : Enters a string or string-type column to trim. If this value is **NULL**, **NULL** is returned.
- [**LEADING** | **TRAILING** | **BOTH**] : You can specify an option to trim a specified string that is in a certain position of the target string. If it is **LEADING**, trimming is performed in front of a character string if it is **TRAILING**, trimming is performed at the back of a character string if it is **BOTH**, trimming is performed in front and at the back of a character string. If the option is not specified, **BOTH** is specified by default.
- The character string of *trim_string* and *string* should have the same character set.

Example

```
--trimming NULL returns NULL
SELECT TRIM (NULL);
 trim(both  from null)
======================
  NULL

--trimming spaces on both leading and trailing parts
SELECT TRIM ('     Olympic     ');
 trim(both  from '     Olympic     ')
```

```
======================
  'Olympic'

--trimming specific strings on both leading and trailing parts
SELECT TRIM ('i' FROM 'iiiiiOlympiciiiii');
 trim(both 'i' from 'iiiiiOlympiciiiii')
======================
  'Olympic'

--trimming specific strings on the leading part
SELECT TRIM (LEADING 'i' FROM 'iiiiiOlympiciiiii');
 trim(leading 'i' from 'iiiiiOlympiciiiii')
======================
  'Olympiciiiii'

--trimming specific strings on the trailing part
SELECT TRIM (TRAILING 'i' FROM 'iiiiiOlympiciiiii');
 trim(trailing 'i' from 'iiiiiOlympiciiiii')
======================
  'iiiiiOlympic'
```

UCASE, UPPER Function

Description

The **UCASE** and **UPPER** functions convert lowercase characters that are included in a character string to uppercase characters. Note that the **UPPER** function may not work properly in character sets that are not supported by CUBRID. For details about the character sets supported by CUBRID, see Definition and Characteristics.

Syntax

```
UCASE ( string )
UPPER ( string )

string :
• character string
• NULL
```

- *string* : Specifies the string in which lowercase characters are to be converted to uppercase. If the value is **NULL**, **NULL** is returned.

Example

```
SELECT UPPER('');
 upper('')
======================
  ''

SELECT UPPER(NULL);
 upper(null)
======================
  NULL

SELECT UPPER('Cubrid');
 upper('Cubrid')
======================
  'CUBRID'
```

Numeric and Operator Functions

ABS Function

Description

The **ABS** function returns the absolute value of a given number. The data type of the return value is the same as that of the argument.

Syntax

```
ABS( number_operand )
```

- *number_operand* : An operator which returns a numeric value

Example

```
--it returns the absolute value of the argument
SELECT ABS(12.3), ABS(-12.3), ABS(-12.3000), ABS(0.0);
  abs(12.3)             abs(-12.3)            abs(-12.3000)         abs(0.0)
=================================================================================
  12.3                  12.3                  12.3000               .0
```

ACOS Function

Description

The **ACOS** function returns an arc cosine value of the argument. That is, it returns a value whose cosine is *x* in radian. The return value is a **DOUBLE** type. x must be a value between -1 and 1, inclusive. Otherwise, **NULL** is returned.

Syntax

```
ACOS( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT ACOS(1), ACOS(0), ACOS(-1);
  acos(1)                   acos(0)                   acos(-1)
===================================================================================
  0.000000000000000e+00     1.570796326794897e+00     3.141592653589793e+00
```

ASIN Function

Description

The **ASIN** function returns an arc sine value of the argument. That is, it returns a value whose sine is *x* in radian. The return value is a **DOUBLE** type. x must be a value between -1 and 1, inclusive. Otherwise, **NULL** is returned.

Syntax

```
ASIN ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT ASIN(1), ASIN(0), ASIN(-1);
  asin(1)                   asin(0)                   asin(-1)
===============================================================================
  1.570796326794897e+00     0.000000000000000e+00    -1.570796326794897e+00
```

ATAN Function

Description

The **ATAN** function returns a value whose tangent is *x* in radian. The argument *y* can be omitted. If *y* is specified, the function calculates the arc tangent value of *y/x*. The return value is a **DOUBLE** type.

Syntax

```
ATAN ( [y,] x )
```

- *x*, *y* : An expression that returns a numeric value.

Example

```
SELECT ATAN(1), ATAN(-1), ATAN(1,-1);
                   atan(1)                   atan(-1)              atan2(1, -1)
==============================================================================
     7.853981633974483e-01    -7.853981633974483e-01     2.356194490192345e+00
```

ATAN2 Function

Description

The **ATAN2** function returns the arc tangent value of *y/x* in radian. This function is working like the [ATAN](#)**Function**. Arguments *x* and *y* must be specified. The return value is a **DOUBLE** type.

Syntax

```
ATAN2 ( y, x )
```

- *x*, *y* : An expression that returns a numeric value.

Example

```
SELECT ATAN2(1,1), ATAN2(-1,-1), ATAN2(Pi(),0);
atan2(1, 1)             atan2(-1, -1)           atan2( pi(), 0)
==============================================================================
 7.853981633974483e-01    -2.356194490192345e+00     1.570796326794897e+00
```

CEIL Function

Description

The **CEIL** function returns the smallest integer that is not less than its argument. The return value is determined based on the valid number of digits that are specified as the *number_operand* argument.

Syntax

```
CEIL( number_operand )
```

- *number_operand* : An expression that returns a numeric value.

Example

```
SELECT CEIL(34567.34567), CEIL(-34567.34567);
  ceil(34567.34567)      ceil(-34567.34567)
============================================
  34568.00000            -34567.00000

SELECT CEIL(34567.1), CEIL(-34567.1);
  ceil(34567.1)          ceil(-34567.1)
=============================
  34568.0          -34567.0
```

COS Function

Description

The **COS** function returns a cosine value of the argument. The argument *x* must be a radian value. The return value is a **DOUBLE** type.

Syntax

```
COS( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT COS(pi()/6), COS(pi()/3), COS(pi());
  cos( pi()/6)              cos( pi()/3)              cos( pi())
==============================================================================
  8.660254037844387e-01     5.000000000000001e-01    -1.000000000000000e+00
```

COT Function

Description

The **COT** function returns the cotangent value of the argument *x*. That is, it returns a value whose tangent is *x* in radian. The return value is a **DOUBLE** type.

Syntax

```
COT ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT COT(1), COT(-1), COT(0);
  cot(1)                   cot(-1)    cot(0)
==========================================================================
  6.420926159343306e-01    -6.420926159343306e-01   NULL
```

DEGREES Function

Description

The **DEGREES** function returns the argument *x* specified in radian converted to a degree value. The return value is a **DOUBLE** type.

Syntax

```
DEGREES ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT DEGREES(pi()/6), DEGREES(pi()/3), DEGREES (pi());
  degrees( pi()/6)          degrees( pi()/3)            degrees( pi())
==============================================================================
  3.000000000000000e+01     5.999999999999999e+01     1.800000000000000e+02
```

DRANDOM/DRAND Function

Description

The **DRANDOM/DRAND** function returns a random double-precision floating point value in the range of between 0.0 and 1.0. A *seed* argument that is **INTEGER** type can be specified. It rounds up real numbers and an error is returned when it exceeds the range of **INTEGER**.

The **DRAND** function performs the operation only once to produce only one random number regardless of the number of rows where the operation is output, but the **DRANDOM** function performs the operation every time the statement is repeated to produce a different random value for each row. Therefore, to output rows in a random order, you must use the **DRANDOM** function in the **ORDER BY** clause.

To obtain a random integer value, use the RANDOM/RAND Function.

Syntax

```
DRANDOM( [seed] )
DRAND( [seed] )
```

Example

```
SELECT DRAND(), DRAND(1), DRAND(1.4);
                   drand()                  drand(1)                drand(1.4)
==============================================================================
    2.849646518006921e-001    4.163034446537495e-002    4.163034446537495e-002

SELECT * FROM rand_tbl;
           id  name
===================================
            1  'a'
            2  'b'
            3  'c'
            4  'd'
            5  'e'
            6  'f'
            7  'g'
            8  'h'
            9  'i'
           10  'j'

--drandom() returns random values on every row
SELECT DRAND(), DRANDOM() FROM rand_tbl;
   drand()                 drandom()
==============================================================================
   7.638782921842098e-001    1.018707846308786e-001
   7.638782921842098e-001    3.191320535905026e-001
   7.638782921842098e-001    3.461714529862361e-001
   7.638782921842098e-001    6.791894283883175e-001
   7.638782921842098e-001    4.533829767754143e-001
   7.638782921842098e-001    1.714224677266762e-001
   7.638782921842098e-001    1.698049867244484e-001
   7.638782921842098e-001    4.507583849604786e-002
   7.638782921842098e-001    5.279091769157994e-001
   7.638782921842098e-001    7.021088290047914e-001

--selecting rows in random order
SELECT * FROM rand_tbl ORDER BY DRANDOM();
           id  name
===================================
            6  'f'
            2  'b'
            7  'g'
            8  'h'
            1  'a'
            4  'd'
           10  'j'
            9  'i'
            5  'e'
            3  'c'
```

EXP Function

Description

The **EXP** function returns e^x (the base of natural logarithm) raised to a power.

Syntax

```
EXP( x )
```

- *x* : An operator which returns a numeric value

Example

```
SELECT EXP(1), EXP(0);
  exp(1)                    exp(0)
====================================================
  2.718281828459045e+000 1.000000000000000e+000

SELECT EXP(-1), EXP(2.00);
  exp(-1)                 exp(2.00)
```

```
====================================================
  3.678794411714423e-001 7.389056098930650e+000
```

FLOOR Function

Description

The **FLOOR** function returns the largest integer that is not greater than its argument. The data type of the return value is the same as that of the argument.

Syntax

```
FLOOR( number_operand )
```

- *number_operand* : An operator which returns a numeric value

Example

```
--it returns the largest integer less than or equal to the arguments
SELECT FLOOR(34567.34567), FLOOR(-34567.34567);

============================================
  34567.00000           -34568.00000

SELECT FLOOR(34567), FLOOR(-34567);
=============================
         34567         -34567
```

FORMAT Function

Description

The **FORMAT** function displays the number *x* by using commas as thousands delimiters, so that its format becomes '#,###,###.#####' and performs rounding after the decimal point to express as many as *dec* digits after it. The return value is a string type.

Syntax

```
FORMAT ( x , dec )
```

- *x* , *dec* : An expression that returns a numeric value.

Example

```
SELECT FORMAT(12000.123456,3), FORMAT(12000.123456,0);
  format(12000.123456, 3)   format(12000.123456, 0)
============================================
  '12,000.123'          '12,000'
```

GREATEST Function

Description

The **GREATEST** function compares more than one expression specified as parameters and returns the greatest value. If only one expression has been specified, the expression is returned because there is no expression to be compared with.

Therefore, more than one expression that are specified as parameters must be of the type that can be compared with each other. If the types of the specified parameters are identical, so are the types of the return values; if they are different, the type of the return value becomes a convertible common data type.

That is, the **GREATEST** function compares the values of column 1, column 2 and column 3 in the same row and returns the greatest value while the **MAX** function compares the values of column in all result rows and returns the greatest value.

Syntax

```
GREATEST( expression [, expression]* )
```

- *expression* : Specifies more than one expression. Their types must be comparable each other. One of the arguments is **NULL**, **NULL** is returned.

Example

The following is an example that returns the number of every medals and the highest number that Korea won. (demodb)

```
SELECT gold, silver , bronze, GREATEST (gold, silver, bronze) FROM participant
WHERE nation_code = 'KOR';
gold       silver       bronze  greatest(gold, silver, bronze)
=======================================================================
           9           12           9                              12
           8           10          10                              10
           7           15           5                              15
          12            5          12                              12
          12           10          11                              12
```

LEAST Function

Description

The **LEAST** function compares more than one expression specified as parameters and returns the smallest value. If only one expression has been specified, the expression is returned because there is no expression to be compared with.

Therefore, more than one expression that are specified as parameters must be of the type that can be compared with each other. If the types of the specified parameters are identical, so are the types of the return values; if they are different, the type of the return value becomes a convertible common data type.

That is, the **LEAST** function compares the values of column 1, column 2 and column 3 in the same row and returns the smallest value while the **MIN** function compares the values of column in all result rows and returns the smallest value.

Syntax

```
LEAST( expression [, expression]* )
```

- *expression* : Specifies more than one expression. Their types must be comparable each other. One of the arguments is **NULL**, **NULL** is returned.

Example

The following is an example that returns the number of every medals and the lowest number that Korea won. (demodb)

```
SELECT gold, silver , bronze, LEAST(gold, silver, bronze) FROM participant
WHERE nation_code = 'KOR';
         gold       silver       bronze  least(gold, silver, bronze)
====================================================================
           9           12           9                           9
           8           10          10                           8
           7           15           5                           5
          12            5          12                           5
          12           10          11                          10
```

LN Function

Description

The **LN** function returns the natural log value (base = e) of an antilogarithm *x*. The return value is a **DOUBLE** type. If the antilogarithm is 0 or a negative number, an error is returned.

Syntax

```
LN ( x )
```

- *x* : An expression that returns a positive value.

Example

```
SELECT ln(1), ln(2.72);
     ln(1)                     ln(2.72)
=====================================================
     0.000000000000000e+00     1.000631880307906e+00
```

LOG2 Function

Description

The **LOG2** function returns a log value whose antilogarithm is *x* and base is 2. The return value is a **DOUBLE** type. If the antilogarithm is 0 or a negative number, an error is returned.

Syntax

```
LOG2 ( x )
```

- *x* : An expression that returns a positive number.
- fails.

Example

```
 SELECT log2(1), log2(8);
     log2(1)                   log2(8)
======================================================
     0.000000000000000e+00     3.000000000000000e+00
```

LOG10 Function

Description

The **LOG10** function returns the common log value of an antilogarithm *x*. The return value is a **DOUBLE** type. If the antilogarithm is 0 or a negative number, an error is returned.

Syntax

```
LOG10 ( x )
```

- *x* : An expression that returns a positive number.

Example

```
SELECT log10(1), log10(1000);
     log10(1)                  log10(1000)
====================================================
     0.000000000000000e+00     3.000000000000000e+00
```

MOD Function

Description

The **MOD** function returns the remainder of the first parameter *m* divided by the second parameter *n*. If *n* is 0, *m* is returned without the division operation being performed.

Note that if the dividend, the parameter m of the **MOD** function, is a negative number, the function operates differently from a typical operation (classical modulus) method.

Result of MOD

m	n	MOD(m, n)	Classical Modulus m-n*FLOOR(m/n)
11	4	3	3
11	-4	3	-1

-11	4	-3	1
-11	-4	-3	-3
11	0	11	Divided by 0 error

Syntax

```
MOD(m, n)
```

- *m* : Represents the dividend. It is an expression that returns a numeric value.
- *n* : Represents the divisor. It is an expression that returns a numeric value.

Example

```
--it returns the reminder of m divided by n
SELECT MOD(11, 4), MOD(11, -4), MOD(-11, 4), MOD(-11, -4), MOD(11,0);
    mod(11, 4)   mod(11, -4)   mod(-11, 4)   mod(-11, -4)   mod(11, 0)
=====================================================================
            3             3           -3             -3           11

SELECT MOD(11.0, 4), MOD(11.000, 4), MOD(11, 4.0), MOD(11, 4.000);
  mod(11.0, 4)          mod(11.000, 4)        mod(11, 4.0)          mod(11, 4.000)
=========================================================================
  3.0                   3.000                 3.0                   3.000
```

PI Function

Description

The **PI** function returns the π value, and is expressed in double-precision.

Syntax

```
PI()
```

Example

```
SELECT PI(), PI()/2;
     pi()                         pi()/2
====================================================
     3.141592653589793e+00     1.570796326794897e+00
```

POW, POWER Function

Description

The **POW** function returns *x* to the power of *y*. **POW** and **POWER** are used interchangeably. The return value is a **DOUBLE** type.

Syntax

```
POW( x, y )
POWER( x, y )
```

- *x* : It represents the base. It is an expression that returns a numeric value. An expression that returns a numeric value.
- *y* : It represents the exponent. An expression that returns a numeric value. If the base is a negative number, an integer must specified as the exponent.

Example

```
SELECT POWER(2, 5), POWER(-2, 5), POWER(0, 0), POWER(1,0);
 power(2, 5)             power(-2, 5)           power(0, 0)           power(1, 0)
======================================================================================
 3.200000000000000e+01    -
3.200000000000000e+01     1.000000000000000e+00     1.000000000000000e+00
```

```
--it returns an error when the negative base is powered by a non-int exponent
SELECT POWER(-2, -5.1), POWER(-2, -5.1);

ERROR
```

RADIANS Function

Description

The **RADIANS** function returns the argument *x* specified in degrees converted to a radian value. The return value is a **DOUBLE** type.

Syntax

```
RADIANS ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT RADIANS(90), RADIANS(180), RADIANS(360);
     radians(90)               radians(180)              radians(360)
==============================================================================
     1.570796326794897e+00     3.141592653589793e+00     6.283185307179586e+00
```

RANDOM/RAND Function

Description

The **RANDOM/RAND** function returns any integer value between 0^2[31] and a *seed* argument that is **INTEGER** type can be specified. It rounds up real numbers and an error is returned when it exceeds the range of **INTEGER**.

The **RAND** function performs the operation only once to produce only one random number regardless of the number of rows where the operation is output, but the **RANDOM** function performs the operation every time the statement is repeated to produce a different random value for each row. Therefore, to output rows in a random order, you must use the **RANDOM** function.

To obtain a random real number, use the DRANDOM/DRAND Function.

Syntax

```
RANDOM( [seed] )
RAND( [seed] )
```

Example

```
SELECT RAND(), RAND(1), RAND(1.4);
       rand()      rand(1)      random(1.4)
====================================================
    1526981144     89400484     89400484

--creating a new table
SELECT * FROM rand_tbl;
           id  name
===================================
            1  'a'
            2  'b'
            3  'c'
            4  'd'
            5  'e'
            6  'f'
            7  'g'
            8  'h'
            9  'i'
           10  'j'

--random() returns random values on every row
SELECT RAND(),RANDOM() FROM rand_tbl;
```

```
       rand()       random()
============================
   2078876566     1753698891
   2078876566     1508854032
   2078876566      625052132
   2078876566      279624236
   2078876566     1449981446
   2078876566     1360529082
   2078876566     1563510619
   2078876566     1598680194
   2078876566     1160177096
   2078876566     2075234419


--selecting rows in random order
SELECT * FROM rand_tbl ORDER BY RANDOM();
           id  name
===================================
            6  'f'
            1  'a'
            5  'e'
            4  'd'
            2  'b'
            7  'g'
           10  'j'
            9  'i'
            3  'c'
            8  'h'
```

ROUND Function

Description

The **ROUND** function returns the specified argument, *number_operand*, rounded to the number of places after the decimal point specified by the *integer*. If the *integer* argument is a negative number, it rounds to a place before the decimal point, that is, at the integer part.

Syntax

```
ROUND( number_operand, integer )
```

- *number_operand* : An expression that returns a numeric value
- *integer* : Specifies the place to round to. If a positive integer *n* is specified, the number is represented to the nth place after the decimal point; if a negative integer *n* is specified, the number is rounded to the *n*th place before the decimal point.
- The return value has the same type as the *number_operand*.

Example

```
--it rounds a number to one decimal point when the second argument is omitted
SELECT ROUND(34567.34567), ROUND(-34567.34567);
  round(34567.34567, 0)   round(-34567.34567, 0)
============================================
  34567.00000           -34567.00000


--it rounds a number to three decimal point
SELECT ROUND(34567.34567, 3), ROUND(-34567.34567, 3)  FROM db_root;
 round(34567.34567, 3)   round(-34567.34567, 3)
============================================
  34567.34600           -34567.34600

--it rounds a number three digit to the left of the decimal point
SELECT ROUND(34567.34567, -3), ROUND(-34567.34567, -3);
 round(34567.34567, -3)   round(-34567.34567, -3)
============================================
  35000.00000           -35000.00000
```

SIGN Function

Description

The **SIGN** function returns the sign of a given number. It returns 1 for a positive value, -1 for a negative value, and 0 for zero.

Syntax

```
SIGN(number_operand)
```

- *number_operand* : An operator which returns a numeric value

Example

```
--it returns the sign of the argument

SELECT SIGN(12.3), SIGN(-12.3), SIGN(0);
    sign(12.3)   sign(-12.3)      sign(0)
========================================
            1            -1            0
```

SIN Function

Description

The **SIN** function returns a sine value of the parameter. The argument *x* must be a radian value. The return value is a **DOUBLE** type.

Syntax

```
SIN ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT SIN(pi()/6), SIN(pi()/3), SIN(pi());
     sin( pi()/6)              sin( pi()/3)              sin( pi())
==============================================================================
     4.999999999999999e-01     8.660254037844386e-01     1.224646799147353e-16
```

SQRT Function

Description

The **SQRT** function returns the square root of *x* as a **DOUBLE** type.

Syntax

```
SORT ( x )
```

- *x* : An expression that returns a numeric value. An error is returned if this value is a negative number.

Example

```
SELECT SQRT(4), SQRT(16.0);
     sqrt(4)                   sqrt(16.0)
====================================================
     2.000000000000000e+00     4.000000000000000e+00
```

TAN Function

Description

The **TAN** function returns a tangent value of the argument. The argument *x* must be a radian value. The return value is a **DOUBLE** type.

Syntax

```
TAN ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT TAN(pi()/6), TAN(pi()/3), TAN(pi()/4);
tan( pi()/6)              tan( pi()/3)              tan( pi()/4)
==============================================================================
     5.773502691896257e-01     1.732050807568877e+00     9.999999999999999e-01
```

TRUNC, TRUNCATE Function

Description

The **TRUNC** and **TRUNCATE** function truncates the numbers of the specified argument *x* to the right of the *dec* position. If the *dec* argument is a negative number, it displays 0s to the *dec*-th position left to the decimal point. Note that the *dec* argument of the **TRUNC** function can be omitted, but that of the **TRUNCATE** function cannot be omitted. If the *dec* argument is a negative number, it displays 0s to the *dec*-th position left to the decimal point. The number of digits of the return value to be represented follows the argument *x*.

Syntax

```
TRUNC( x[, dec] )
TRUNCATE( x, dec )
```

- *x* : An expression that returns a numeric value.
- *dec* : The place to be truncated is specified. If a positive integer *n* is specified, the number is represented to the *n*-th place after the decimal point; if a negative integer *n* is specified, the number is truncated to the *n*-th place before the decimal point. It truncates to the first place after the decimal point if the *dec* argument is 0 or omitted. Note that the *dec* argument cannot be omitted in the **TRUNCATE** function.

Example

```
--it returns a number truncated to 0 places
SELECT TRUNC(34567.34567), TRUNCATE(34567.34567, 0);
  trunc(34567.34567, 0)   trunc(34567.34567, 0)
============================================
  34567.00000            34567.00000

--it returns a number truncated to three decimal places
SELECT TRUNC(34567.34567, 3), TRUNC(-34567.34567, 3);
  trunc(34567.34567, 3)   trunc(-34567.34567, 3)
============================================
  34567.34500           -34567.34500

--it returns a number truncated to three digits left of the decimal point
SELECT TRUNC(34567.34567, -3), TRUNC(-34567.34567, -3);
  trunc(34567.34567, -3)   trunc(-34567.34567, -3)
============================================
  34000.00000           -34000.00000
```

Date/Time Functions and Operators

ADDDATE, DATE_ADD Function

Description

The **ADDDATE** function performs an addition or subtraction operation on a specific **DATE** value; **ADDDATE** and **DATE_ADD** are used interchangeably. They return the DATETIME type in the following cases: 1) The first argument is **DATETIME** or **TIMESTAMP** type, 2) The first argument is **DATE** type, or 3) Less than **DAY** unit is specified for the value of INTERVAL.

Therefore, to return value of **DATETIME** type, you should convert the value of first argument by using the **CAST** function. Even though the date resulting from the operation exceeds the last day of the month, the function returns a valid **DATE** value considering the last date of the month.

Syntax

```
ADDDATE(date, INTERVAL expr unit)
DATE_ADD(date, INTERVAL expr unit)
ADDDATE(date, days)
```

- *date* : It is a **DATE**, **TIMETIME**, or **TIMESTAMP** expression that represents the start date. If an invalid **DATE** value such as '2006-07-00' is specified, an error is returned.
- *expr* : It represents the interval value to be added to the start date. If a negative number is specified next to the **INTERVAL** keyword, the interval value is subtracted from the start date.
- *unit* : It represents the unit of the interval value specified in the *expr* expression. See the following table to specify the format for the interpretation of the interval value. If the value of *expr* is less than the number requested in the *unit*, it is specified from the smallest unit. For example, if it is HOUR_SECOND, three values such as 'HOURS:MINUTES:SECONDS' are required. In the case, if only two values such as "1:1" are given, it is regarded as 'MINUTES:SECONDS'.

expr value for unix

Unit Value	**expr Value**
MILLISECOND	MILLISECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MILLISECOND	'SECONDS.MILLISECONDS'
MINUTE_MILLISECOND	'MINUTES:SECONDS.MILLISECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MILLISECOND	'HOURS:MINUTES:SECONDS.MILLISECONDS'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MILLISECOND	'DAYS HOURS:MINUTES:SECONDS.MILLISECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'

DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

Example

```
SELECT SYSDATE, ADDDATE(SYSDATE,INTERVAL 24 HOUR), ADDDATE(SYSDATE, 1);

   SYS_DATE    date_add( SYS_DATE , INTERVAL 24 HOUR)   adddate( SYS_DATE , 1)
==============================================================================
  03/30/2010  12:00:00.000 AM 03/31/2010              03/31/2010

--it substracts days when argument < 0
SELECT SYSDATE, ADDDATE(SYSDATE,INTERVAL -24 HOUR), ADDDATE(SYSDATE, -1);

   SYS_DATE    date_add( SYS_DATE , INTERVAL -24 HOUR)   adddate( SYS_DATE , -1)
==============================================================================
  03/30/2010  12:00:00.000 AM 03/29/2010              03/29/2010

--when expr is not fully specified for unit
select sys_datetime, adddate(sys_datetime, interval '1:20' HOUR_SECOND);

   SYS_DATETIME                  date_add( SYS_DATETIME , INTERVAL '1:20' HOUR_SECOND)
=======================================================================================
  06:18:24.149 PM 06/28/2010     06:19:44.149 PM 06/28/2010
```

ADD_MONTH Function

The **ADD_MONTHS** function adds a *month* value to the expression *date_parameter* of **DATE** type, and it returns a **DATE** type value. If the day (*dd*) of the value specified as the parameter exists within the month of the result value of the operation, it returns the given day (*dd*); otherwise returns the last day of the given month (*dd*). If the result value of the operation exceeds the expression range of the **DATE** type, it returns an error.

Syntax

```
ADD_MONTHS ( date_argument , month )

date_argument :
• date
• NULL

month :
• integer
• NULL
```

- *date_argument* : Specifies an expression of **DATE** type. To specify a **TIMESTAMP** or **DATETIME** value, an explicit casting to **DATE** type is required. If the value is **NULL**, **NULL** is returned.
- *month* : Specifies the number of the months to be added to the *date_argument*. Both positive and negative values can be specified. If the given value is not an integer type, conversion to an integer type by an implicit casting (rounding to the first place after the decimal point) is performed. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns DATE type value by adding month to the first argument

SELECT ADD_MONTHS(DATE '2008-12-25', 5), ADD_MONTHS(DATE '2008-12-25', -5);
  add_months(date '2008-12-25', 5)   add_months(date '2008-12-25', -5)
=======================================================================
  05/25/2009                         07/25/2008


SELECT ADD_MONTHS(DATE '2008-12-31', 5.5), ADD_MONTHS(DATE '2008-12-31', -5.5);
  add_months(date '2008-12-31', 5.5)   add_months(date '2008-12-31', -5.5)
===========================================================================
  06/30/2009                           06/30/2008

SELECT ADD_MONTHS(CAST (SYS_DATETIME AS DATE), 5), ADD_MONTHS(CAST (SYS_TIMESTAMP AS DATE),
5);
```

```
  add_months( cast( SYS_DATETIME  as date), 5)   add_months( cast( SYS_TIMESTAMP  as date), 5)
================================================================================
  07/03/2010                                    07/03/2010
```

CURDATE, CURRENT_DATE, CURRENT_DATE(), SYS_DATE, SYSDATE

Description

CURDATE(), **CURRENT_DATE**, **CURRENT_DATE**, **SYS_DATE**, and **SYSDATE** are used interchangeably, and they return the current date as the **DATE** type (*MM*/*DD*/*YYYY* or *YYYY-MM-DD*). The unit is day.

Syntax

```
CURDATE()
CURRENT_DATE()
CURRENT_DATE
SYS_DATE
SYSDATE
```

Example

```
--it returns the current date in DATE type
SELECT CURDATE(), CURRENT_DATE(), CURRENT_DATE, SYS_DATE, SYSDATE;
   SYS_DATE    SYS_DATE    SYS_DATE    SYS_DATE    SYS_DATE
============================================================
  04/01/2010  04/01/2010  04/01/2010  04/01/2010  04/01/2010

--it returns the date 60 days added to the current date
SELECT CURDATE()+60;
   SYS_DATE +60
===============
   05/31/2010
```

CURRENT_DATETIME, CURRENT_DATETIME(), NOW(), SYS_DATETIME, SYSDATETIME

Description

CURRENT_DATETIME, **CURRENT_DATETIME**(), **NOW**() **SYS_DATETIME**, and **SYSDATETIME** are used interchangeably, and they return the current date and time in **DATETIME** type. The unit is millisecond.

Syntax

```
CURRENT_DATETIME
CURRENT_DATETIME()
NOW()
SYS_DATETIME
SYSDATETIME
```

Example

```
--it returns the current date and time in DATETIME type
SELECT NOW(), SYS_DATETIME;

   SYS_DATETIME                   SYS_DATETIME
==============================================================
  04:08:09.829 PM 02/04/2010     04:08:09.829 PM 02/04/2010

--it returns the timestamp value 1 hour added to the current sys_datetime value
SELECT TO_CHAR(SYSDATETIME+3600*1000, 'YYYY-MM-DD HH:MI');
  to_char( SYS_DATETIME +3600*1000, 'YYYY-MM-DD HH:MI', 'en_US')
======================
  '2010-02-04 04:08'
```

CURTIME(), CURRENT_TIME, CURRENT_TIME(), SYS_TIME, SYSTIME

Description

CURTIME(), **CURRENT_TIME**, **CURRENT_TIME**(), **SYS_TIME**, and **SYSTIME** are used interchangeably, and they return the current time as the **TIME** type (*HH*:*MI*:SS). The unit is second.

Syntax

```
CURTIME()
CURRENT_TIME
CURRENT_TIME()
SYS_TIME
SYSTIME
```

Example

```
--it returns the current time in TIME type
SELECT CURTIME(), CURRENT_TIME(), CURRENT_TIME, SYS_TIME, SYSTIME;
   SYS_TIME     SYS_TIME     SYS_TIME     SYS_TIME     SYS_TIME
=================================================================
  04:37:34 PM  04:37:34 PM  04:37:34 PM  04:37:34 PM  04:37:34 PM

--it returns the time value 1 hour added to the current sys_time
SELECT CURTIME()+3600;
   SYS_TIME +3600
=================
   05:37:34 PM
```

CURRENT_TIMESTAMP, CURRENT_TIMESTAMP(), SYS_TIMESTAMP, SYSTIMESTAMP, LOCALTIME, LOCATIME(), LOCALTIMESTAMP, LOCALTIMESTAMP()

Description

CURRENT_TIMESTAMP, **CURRENT_TIMESTAMP**(), **SYS_TIMESTAMP**, **SYSTIMESTAMP**, **LOCALTIME**, **LOCALTIME**(), **LOCALTIMESTAMP**, and **LOCALTIMESTAMP**() are used interchangeably, and they return the current date and time as the **TIMESTAMP** type. The unit is second.

If you define **DEFAULT** value for column initial value and specify the initial value to **SYS_DATETIME**, the default value is specified to the timestamp at the time of creating a table, not inserting a table. Note that the default value is not specified in case of INSERT. Therefore, you must specify **SYS_DATETIME** in the **VALUES** of **INSERT** statement upon inserting data.

Syntax

```
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP()
SYS_TIMESTAMP
SYSTIMESTAMP
LOCALTIME
LOCALTIME()
LOCALTIMESTAMP
LOCALTIMESTAMP()
```

Example

```
--it returns the current date and time in TIMESTAMP type
SELECT LOCALTIME, SYS_TIMESTAMP;
 SYS_TIMESTAMP              SYS_TIMESTAMP
==============================================================================
  07:00:48 PM 04/01/2010     07:00:48 PM 04/01/2010

--it returns the timestamp value 1 hour added to the current sys_timestamp value
SELECT CURRENT_TIMESTAMP()+3600;
 SYS_TIMESTAMP +3600
===========================
  08:02:42 PM 04/01/2010
```

DATE Function

Description

The **DATE** function extracts the date part from the specified argument, and returns it as *MM/DD/YYYY*' format string. Arguments that can be specified are **DATE**, **TIMESTAMP** and **DATETIME** types. The return value is a **VARCHAR** type.

Syntax

```
DATE(date)
```

- *date* : The **DATE**, **TIMESTAMP** or **DATETIME** can be specified.

Example

```
SELECT DATE('2010-02-27 15:10:23');
 date('2010-02-27 15:10:23')
======================
  '02/27/2010'

SELECT DATE(NOW());
 date( SYS_DATETIME )
======================
  '04/01/2010'
```

DATEDIFF Function

Description

The **DATEDIFF** function returns the difference between two arguments as an integer representing the number of days. Arguments that can be specified are **DATE**, **TIMESTAMP** and **DATETIME** types and it return value is only **INTEGER** type.

Syntax

```
DATEDIFF (date1,date2)
```

- *date1*, *date2* : The **DATE**, **TIMESTAMP** or **DATETIME** type or date/time format string can be specified. If invalid string is specified, an error is returned.

Example

```
SELECT DATEDIFF('2010-2-28 23:59:59','2010-03-02');
 datediff('2010-2-28 23:59:59', '2010-03-02')
==============================================
                                            -2

SELECT DATEDIFF('2010/12/31', SYSDATETIME);

ERROR: Conversion error in date format.
```

DATE_SUB(), SUBDATE()

Description

DATE_SUB and **SUBDATE**() are used interchangeably, and they perform an addition or subtraction operation on a specific **DATE** value. The return value is a **DATE** or **DATETIME** type. If the date resulting from the operation exceeds the last day of the month, the function returns a valid **DATE** value considering the last date of the month.

Syntax

```
DATE_SUB (date, INTERVAL expr unit)
SUBDATE(date, INTERVAL expr unit)
SUBDATE(date, days)
```

- *date* : It is a **DATE** or **TIMESTAMP** expression that represents the start date. If an invalid **DATE** value such as '2006-07-00' is specified, **NULL** is returned.
- *expr* : It represents the interval value to be subtracted from the start date. If a negative number is specified next to the **INTERVAL** keyword, the interval value is added to the start date.
- *unit* : It represents the unit of the interval value specified in the *exp* expression. To check the expr argument for the unit value, see the table of ADDDATE, DATE_ADD Function.

Example

```
SELECT SYSDATE, SUBDATE(SYSDATE,INTERVAL 24 HOUR), SUBDATE(SYSDATE, 1);
   SYS_DATE     date_sub( SYS_DATE , INTERVAL 24 HOUR)    subdate( SYS_DATE , 1)
==============================================================================
  03/30/2010  12:00:00.000 AM 03/29/2010               03/29/2010

--it adds days when argument < 0
SELECT SYSDATE, SUBDATE(SYSDATE,INTERVAL -24 HOUR), SUBDATE(SYSDATE, -1);
   SYS_DATE     date_sub( SYS_DATE , INTERVAL -24 HOUR)    subdate( SYS_DATE , -1)
==============================================================================
  03/30/2010  12:00:00.000 AM 03/31/2010               03/31/2010
```

DAY/DAYOFMONTH Function

Description

The **DAY** function and the **DAYOFMONTH** function return a day in the ragne of 1 to 31 from the specified parameter. You can specify the **DATE**, **TIMESTAMP**, and **DATETIME** types as parameters and an **INTEGER** will be returned.

Syntax

```
DAY(date)
DAYOFMONTH(date)
```

- *date* : Date

Example

```
SELECT DAYOFMONTH('2010-09-09');
   dayofmonth('2010-09-09')
===========================
                          9

SELECT DAY('2010-09-09 19:49:29');
   day('2010-09-09 19:49:29')
=============================
                            9

SELECT DAYOFMONTH('01:02:03');
ERROR: Conversion error in date format.
```

DAYOFWEEK Function

Description

The **DAYOFWEEK** function returns a day in the range of 1 to 7 (1: Sunday, 2: Monday, ..., 7: Saturday) from the specified parameters. The day index is same as the ODBC standards. You can specify the **DATE**, **TIMESTAMP**, and **DATETIME** types as parameters and an **INTEGER** will be returned.

Syntax

```
DAYOFWEEK(date)
```

- *date* : Date

Example

```
SELECT DAYOFWEEK('2010-09-09');
   dayofweek('2010-09-09')
```

```
==========================
                         5

SELECT DAYOFWEEK('2010-09-09 19:49:29');
 dayofweek('2010-09-09 19:49:29')
=================================
                                5

SELECT DAYOFWEEK('10:28:00');

ERROR: Conversion error in date format.
```

DAYOFYEAR Function

Description

The **DAYOFYEAR** function returns the day of a year in the range of 1 to 366. You can specify the **DATE**, **TIMESTAMP**, and **DATETIME** types as parameters and an **INTEGER** will be returned.

Syntax

```
DAYOFYEAR(date)
```

- *date* : Date

Example

```
SELECT DAYOFYEAR('2010-09-09');
   dayofyear('2010-09-09')
==========================
                       252

SELECT DAYOFYEAR('2010-09-09 19:49:29');
dayofyear('2010-09-09 19:49:29')
=================================
                            252

SELECT DAYOFYEAR('10:28:00');

ERROR: Conversion error in date format.
```

EXTRACT Operator

Description

The **EXTRACT** operator extracts the values from *date-time_argument* and then converts the value type into **INTEGER**.

Syntax

```
EXTRACT ( field FROM date-time_argument )

field :
• YEAR
• MONTH
• DAY
• HOUR
• MINUTE
• SECOND
• MILLISECOND
```

- *field* : Specifies a value to be extracted from date-time expression.
- *date-time argument* : An expression that returns a value of date-time. This expression must be one of **TIME**, **DATE**, **TIMESTAMP**, or **DATETIME** types. If the value is **NULL**, **NULL** is returned.

Example

```
SELECT EXTRACT(MONTH FROM DATETIME '2008-12-25 10:30:20.123' );
  extract(month  from datetime '2008-12-25 10:30:20.123')
```

```
=========================================================
                                                       12

SELECT EXTRACT(HOUR FROM DATETIME '2008-12-25 10:30:20.123' );
 extract(hour  from datetime '2008-12-25 10:30:20.123')
=========================================================
                                                       10

SELECT EXTRACT(MILLISECOND FROM DATETIME '2008-12-25 10:30:20.123' );
 extract(millisecond  from datetime '2008-12-25 10:30:20.123')
=========================================================
                                                      123
```

FROM_DAYS Function

Description

The **FROM_DAYS** function returns a date from the specified parameter. You can specify an **INTEGER** type in the range of 366 to 3652424 as a parameter and a **DATE** type will be returned.

It is not recommended to use the **FROM_DAYS** function for dates prior to the year 1582, as the function does not take dates prior to the introduction of the Gregorian Calendar into account.

Syntax

```
FROM_DAYS(N)
```

- *N* : Integer in the range of 366 to 3,652,424. The maximum value of 3,652,424 means the last day of the year 9999.

Example

```
SELECT FROM_DAYS(719528);
   from_days(719528)
====================
  01/01/1970

SELECT FROM_DAYS('366');
  from_days('366')
=================
  01/03/1

SELECT FROM_DAYS(3652424);
   from_days(3652424)
=====================
  12/31/9999

SELECT FROM_DAYS(3652425);
ERROR: Conversion error in date format.

SELECT FROM_DAYS(-1);
ERROR: Conversion error in date format.
```

FROM_UNIXTIME Function

Description

The **FROM_UNIXTIME** function returns the date and time in the format of 'YYYY-MM-DD HH:MM:SS.' You can enter an **INTEGER** type that corresponds to the UNIX timestamp and a **VARCHAR** type will be returned. The return value will be displayed in the current time zone.

Displays the the result according to the format that you specified, and the time *format* format follows the date/time format 2 of DATE_FORMAT Function.

The relationship is not one of one-to-one correspondence between the **TIMESTAMP** and the UNIX timestamp so if you use the **UNIX_TIMESTAMP** function or the **FROM_UNIXTIME** function, partial value could be lost. For more information, see UNIX_TIMESTAMP Function.

Syntax

```
FROM_UNIXTIME( unix_timestamp[, format] )
```

- *unix_timestamp* : Positive integer
- *format* : Time format. Follows the date/time format of the DATE_FORMAT Function.

Example

```
SELECT FROM_UNIXTIME(1234567890);
   from_unixtime(1234567890)
============================
  01:31:30 AM 02/14/2009

SELECT FROM_UNIXTIME('1000000000');
   from_unixtime('1000000000')
==============================
  04:46:40 AM 09/09/2001

SELECT FROM_UNIXTIME(1234567890,'%M %Y %W');
   from_unixtime(1234567890, '%M %Y %W')
======================
  'February 2009 Saturday'

SELECT FROM_UNIXTIME('1234567890','%M %Y %W');
   from_unixtime('1234567890', '%M %Y %W')
======================
  'February 2009 Saturday'

SELECT FROM_UNIXTIME(-1);
ERROR: Conversion error in timestamp format.
Download in other formats:
```

LAST_DAY Function

Description

The **LAST_DAY** function returns the last day of the given month as a **DATE** type.

Syntax

```
LAST_DAY ( date_argument )

date_argument :
• date
• NULL
```

- *date_argument* : Specifies an expression of **DATE** type. To specify a **TIMESTAMP**or **DATETIME** value, explicit casting to **DATE** is required. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns last day of the momth in DATE type
SELECT LAST_DAY(DATE '1980-02-01'), LAST_DAY(DATE '2010-02-01');
  last_day(date '1980-02-01')    last_day(date '2010-02-01')
============================================================
  02/28/1980                   02/28/2010

--it returns last day of the momth when explicitly casted to DATE type
SELECT LAST_DAY(CAST (SYS_TIMESTAMP AS DATE)), LAST_DAY(CAST (SYS_DATETIME AS DATE));
  last_day( cast( SYS_TIMESTAMP  as date))    last_day( cast( SYS_DATETIME  as date))
================================================================================
  02/28/2010                                 02/28/2010
```

MAKEDATE Function

Description

The **MAKEDATE** function returns a date from the specified parameter. You can specify an **INTEGER** type corresponding to the day of the year in the the range of 1 to 9999 as a parameter, and the **DATE** type will be returned in the range of 1/1/1 to 12/31/9999. If the day of the year has passed the corresponding year, it will become the next year. For example, MAKEDATE(1999, 366) will return 2000-01-01.

However, if you input a value in the range of 0 to 69 as the year, it will be processed as the year 2000-2069, if it is a value in the range of 70 to 99, it will be processed as the year 1970-1999.

Syntax

```
MAKEDATE(year, dayofyear)
```

- *year* : Year in the range of 1 to 9999
- *dayofyear* : If you input a value in the range of 0 to 99 as the year, only the year after 100 years will be used. Therefore, the maximum value of *dayofyear* is 3,615,902 and MAKEDATE(100, 3615902) returns 9999/12/31.

Example

```
SELECT MAKEDATE(2010,277);
   makedate(2010, 277)
======================
  10/04/2010

SELECT MAKEDATE(10,277);
   makedate(10, 277)
====================
  10/04/2010

SELECT MAKEDATE(70,277);
   makedate(70, 277)
====================
  10/04/1970

SELECT MAKEDATE(100,3615902);
   makedate(100, 3615902)
=========================
  12/31/9999

SELECT MAKEDATE('9999','365');
   makedate('9999', '365')
======================
  12/31/9999

SELECT MAKEDATE(9999,366);

ERROR: Conversion error in date format.
```

MAKETIME Function

Description

The **MAKETIME** function returns the hour from the specified parameter in the AM/PM format. You can specify the **INTEGER** types corresponding hours, minutes and seconds as parameters and a **DATETIME** type will be returned.

Syntax

```
MAKETIME(hour, min, sec)
```

- *hour* : Integers representing the hours in the range of 0 to 23
- *min* : Integers representing the minutes in the range of 0 to 59
- *sec* : Integers representing the seconds in the range of 0 to 59

Example

```
SELECT MAKETIME(13,34,4);
   maketime(13, 34, 4)
======================
  01:34:04 PM

SELECT MAKETIME('1','34','4');
   maketime('1', '34', '4')
===========================
  01:34:04 AM

SELECT MAKETIME(24,0,0);

ERROR: Conversion error in time format.
```

MINUTE Function

Description

The **MINUTE** function returns the minutes in the range of 0 to 59 from the specified parameter. You can specify the **TIME**, **TIMESTAMP**, **DATETIME** types as parameters and an **INTEGER** type will be returned.

Syntax

```
MINUTE(time)
```

- *time* : Time

Example

```
SELECT MINUTE('12:34:56');
   minute('12:34:56')
=====================
                   34

SELECT MINUTE('2010-01-01 12:34:56');
   minute('2010-01-01 12:34:56')
================================
                              34

SELECT MINUTE('2010-01-01 12:34:56.7890');
   minute('2010-01-01 12:34:56.7890')
=====================================
                                   34

SELECT MINUTE('2010-01-01');

  In the command from line 1,
  ERROR: Conversion error in time format.
```

MONTH Function

Description

The MONTH function returns the month in the range of 1 to 12 from the specified parameter. You can specify the **DATE**, **TIMESTAMP**, and **DATETIME** types as parameters and an **INTEGER** type will be returned.

Syntax

```
MONTH(date)
```

- *date* : Date

Example

```
SELECT MONTH('2010-01-02');
   month('2010-01-02')
======================
```

```
                     1

SELECT MONTH('2010-01-02 12:34:56');
   month('2010-01-02 12:34:56')
===============================
                              1

SELECT MONTH('2010-01-02 12:34:56.7890');
   month('2010-01-02 12:34:56.7890')
====================================
                                   1

SELECT MONTH ('12:34:56');

ERROR: Conversion error in date format.
```

MONTHS_BETWEEN Function

Description

The **MONTHS_BETWEEN** function returns the difference between the given **DATE** value. The return value is **DOUBLE** type. An integer value is returned if the two dates specified as parameters are identical or are the last day of the given month; otherwise, a value obtained by dividing the day difference by 31 is returned.

Syntax

```
MONTHS_BETWEEN(date_argument, date_argument)

date_argument :
• date
• NULL
```

- *date_argument* : Specifies an expression of **DATE** type. To specify a **TIMESTAMP**or **DATETIME** value, explicit casting to **DATE** is required. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns the negative months when the first argument is the previous date
SELECT MONTHS_BETWEEN(DATE '2008-12-31', DATE '2010-6-30');
 months_between(date '2008-12-31', date '2010-6-30')
======================================================
                               -1.800000000000000e+001

--it returns integer values when each date is the last dat of the month
SELECT MONTHS_BETWEEN(DATE '2010-6-30', DATE '2008-12-31');
 months_between(date '2010-6-30', date '2008-12-31')
======================================================
                                1.800000000000000e+001

--it returns months between two arguments when explicitly casted to DATE type
SELECT MONTHS_BETWEEN(CAST (SYS_TIMESTAMP AS DATE), DATE '2008-12-25');
 months_between( cast( SYS_TIMESTAMP  as date), date '2008-12-25')
====================================================================
                                              1.332258064516129e+001

--it returns months between two arguments when explicitly casted to DATE type
SELECT MONTHS_BETWEEN(CAST (SYS_DATETIME AS DATE), DATE '2008-12-25');
 months_between( cast( SYS_DATETIME  as date), date '2008-12-25')
===================================================================
                                             1.332258064516129e+001
```

QUARTER Function

Description

The QUARTER function returns the quarter in the range of 1 to 4 from the specified parameter. You can specify the **DATE**, **TIMESTAMP**, and **DATETIME** types as parameters and an **INTEGER** type will be returned.

Syntax

```
QUARTER(date)
```

- *date* : Date

Example

```
SELECT QUARTER('2010-05-05');
   quarter('2010-05-05')
========================
                       2

SELECT QUARTER('2010-05-05 12:34:56');
  quarter('2010-05-05 12:34:56')
===============================
                              2

SELECT QUARTER('2010-05-05 12:34:56.7890');
  quarter('2010-05-05 12:34:56.7890')
==================================
                              2

SELECT QUARTER('12:34:56');

ERROR: Conversion error in date format.
```

SEC_TO_TIME Function

Description

The **SEC_TO_TIME** function returns the time including hours, minutes and seconds from the specified parameters. You can specify the **INTEGER** type in the range of 0 to 86,399 as a parameter and the TIME type will be returned.

Syntax

```
SEC_TO_TIME(second)
```

- *second* : Seconds in the range of 0 to 86,399

Example

```
SELECT SEC_TO_TIME(82800);
   sec_to_time(82800)
=====================
  11:00:00 PM

SELECT SEC_TO_TIME('82800.3');
   sec_to_time('82800.3')
=========================
  11:00:00 PM

SELECT SEC_TO_TIME(86399)
   sec_to_time(86399)
=====================
  11:59:59 PM

SELECT SEC_TO_TIME(86400);

ERROR: Conversion error in time format.
```

SECOND Function

Description

The **SECOND** function returns the seconds in the range of 0 to 59 from the specified parameter. You can specify the **TIME**, **TIMESTAMP**, and **DATETIME** types as parameters, and an **INTEGER** type will be returned.

If the function fails, **NULL** is returned when the database server configuration parameter **return_null_on_function_errors** is set to yes. When the parameter is set to no, the function outputs error message. The default value of **return_null_on_function_errors** is **no**.

Syntax

```
SECOND(time)
```

- *time* : Time

Example

```
SELECT SECOND('12:34:56');
   second('12:34:56')
=====================
                   56

SELECT SECOND('2010-01-01 12:34:56');
   second('2010-01-01 12:34:56')
================================
                              56

SELECT SECOND('2010-01-01 12:34:56.7890');
   second('2010-01-01 12:34:56.7890')
=====================================
                                   56

SELECT SECOND ('2010-01-01');

ERROR: Conversion error in time format.
```

STR_TO_DATE Function

Description

The **STR_TO_DATE** function converts the given character string to a date/time value by interpreting it according to the specified format and operates in the opposite way to the DATE_FORMAT Function. The return value is determined depending on the date/time part included in the character string and it is one of the **DATETIME**, **DATE**, TIME types.

If the *string* includes an invalid date/time value. or the character string can nott be interpreted by applying the format specifier specified in the *format*, an error will be returned.

Syntax

```
STR_TO_DATE(string, format)
```

- *string* : All character string types can be specified.
- *format* : Specifies the format to interpret the character string. You should use character strings including % for the format specifiers. See the table, date/time format 2 of DATE_FORMAT Function.

Example

```
SELECT STR_TO_DATE('01,5,2013','%d,%m,%Y');
 str_to_date('01,5,2013', '%d,%m,%Y')
=======================================
  05/01/2013

SELECT STR_TO_DATE('May 1, 2013','%M %d,%Y');
 str_to_date('May 1, 2013', '%M %d,%Y')
=========================================
  05/01/2013

SELECT STR_TO_DATE('a09:30:17','a%h:%i:%s');
 str_to_date('a09:30:17', 'a%h:%i:%s')
========================================
  09:30:17 AM

SELECT STR_TO_DATE('09:30:17a','%h:%i:%s');
 str_to_date('09:30:17a', '%h:%i:%s')
```

```
=======================================
  09:30:17 AM
```

CURTIME(), CURRENT_TIME, CURRENT_TIME(), SYS_TIME, SYSTIME

Description

CURTIME(), **CURRENT_TIME**, **CURRENT_TIME**(), **SYS_TIME**, and **SYSTIME** are used interchangeably, and they return the current time as the **TIME** type (*HH*:*MI*:SS). The unit is second.

Syntax

```
CURTIME()
CURRENT_TIME
CURRENT_TIME()
SYS_TIME
SYSTIME
```

Example

```
--it returns the current time in TIME type
SELECT CURTIME(), CURRENT_TIME(), CURRENT_TIME, SYS_TIME, SYSTIME;
   SYS_TIME     SYS_TIME     SYS_TIME     SYS_TIME     SYS_TIME
=================================================================
  04:37:34 PM  04:37:34 PM  04:37:34 PM  04:37:34 PM  04:37:34 PM

--it returns the time value 1 hour added to the current sys_time
SELECT CURTIME()+3600;
   SYS_TIME +3600
=================
   05:37:34 PM
```

TIME_TO_SEC Function

Description

The TIME_TO_SEC function returns the seconds in the range of 0 to 86,399 from the specified parameters. You can specify the **TIME**, **TIMESTAMP**, and **DATETIME** types as parameters, and an **INTEGER** type will be returned.

Syntax

```
TIME_TO_SEC(time)
```

- *time* : Time

Example

```
SELECT TIME_TO_SEC('23:00:00');
   time_to_sec('23:00:00')
==========================
                     82800

SELECT TIME_TO_SEC('2010-10-04 23:00:00');
   time_to_sec('2010-10-04 23:00:00')
=====================================
                                82800

 SELECT TIME_TO_SEC('2010-10-04 23:00:00.1234');
   time_to_sec('2010-10-04 23:00:00.1234')
==========================================
                                     82800

SELECT TIME_TO_SEC('2010-01-01');

ERROR: Conversion error in time format.
```

TIMEDIFF Function

Description

The TIMEDIFF function returns the time difference between the two specified time parameters.

You can enter a date/time type, the **TIME**, **DATE**, **TIMESTAMP**, **DATETIME** types as parameters and the data types of the two parameters must be identical. The **TIME** will be returned and the time difference between the two parameters must be in the range of 00:00:00 -23:59:59. If it exceeds the range, an error will be returned.

Syntax

```
TIMEDIFF(expr1, expr2)
```

- *expr1*, *expr2* : Time. The data types of the two parameters must be identical.

Example

```
SELECT TIMEDIFF(time '17:18:19', time '12:05:52');
   timediff(time '17:18:19', time '12:05:52')
=============================================
  05:12:27 AM

SELECT TIMEDIFF('17:18:19','12:05:52');
   timediff('17:18:19', '12:05:52')
===================================
  05:12:27 AM

SELECT TIMEDIFF('2010-01-01 06:53:45', '2010-01-01 03:04:05');
   timediff('2010-01-01 06:53:45', '2010-01-01 03:04:05')
=========================================================
  03:49:40 AM

SELECT TIMEDIFF('2010-01-02 06:53:45', '2010-01-01 03:04:05');

ERROR: ERROR: Conversion error in time format.
```

TIMESTAMP Function

Description

The **TIMESTAMP** function converts a **DATE** or **TIMESTAMP** type expression to a **DATETIME** type.

If the **DATA** format string ('YYYY-MM-DD' or 'MM/DD/YYYY) or **TIMESTAMP** format string ('YYYY-MM-DD HH:MI:SS' or 'HH:MI:SS MM/DD/ YYYY') is specified as the first argument, the function returns it as **DATETIME**.

If the **TIME** format string ('HH:MI:SS') is specified as the second, the function adds it to the first argument and returns the result as a **DATETIME** type. If the second argument is not specified, 12:00:00.000 AM is specified by default.

Syntax

```
TIMESTAMP(date [,time])
```

- *date* : The **DATE** or **TIMESTAMP** type string can be specified.
- *time* : The **TIME** type (HH:MI:SS) can be specified.

Example

```
SELECT TIMESTAMP('2009-12-31'), TIMESTAMP('2009-12-31','12:00:00');
=====================================================================
  12:00:00.000 AM 12/31/2009     12:00:00.000 PM 12/31/2009

SELECT TIMESTAMP('2010-12-31 12:00:00','12:00:00');
===============================================
  12:00:00.000 AM 01/01/2011

SELECT TIMESTAMP('13:10:30 12/25/2008');
===================================
  01:10:30.000 PM 12/25/2008
```

TO_DAYS Function

Description

The **TO_DAYS** function returns the number of days after year 0 in the rage of 366 to 3652424 from the specified parameters. You can specify **DATE** type as a parameter and an **INTEGER** type will be returned.

It is not recommended to use the **TO_DAYS** function for dates prior to the year 1582, as the function does not take dates prior to the introduction of the Gregorian Calendar into account.

Syntax

```
TO_DAYS(date)
```

- *date* : Date

Example

```
SELECT TO_DAYS('2010-10-04');
   to_days('2010-10-04')
========================
                  734414

SELECT TO_DAYS('2010-10-04 12:34:56');
   to_days('2010-10-04 12:34:56')
================================
                          734414

SELECT TO_DAYS('2010-10-04 12:34:56.7890');
   to_days('2010-10-04 12:34:56.7890')
======================================
                                734414

SELECT TO_DAYS('1-1-1');
   to_days('1-1-1')
===================
                366

SELECT TO_DAYS('9999-12-31');
   to_days('9999-12-31')
========================
                 3652424

SELECT TO_DAYS ('12:34:56');

ERROR: Conversion error in date format.
```

UNIX_TIMESTAMP Function

Description

The arguments of the **UNIX_TIMESTAMP** function can be omitted. If they are omitted, the function returns the interval between '1970-01-01 00:00:00' UTC and the current system date/time in seconds as a **INTEGER** value. If the date argument is specified, the function returns the interval between '1970-01-01 00:00:00' UTC and the specified date/time in seconds.

Syntax

```
UNIX_TIMESTAMP( [date] )
```

- *date* : **DATE** or **TIMESTAMP** type strings, or strings in the '*YYYYMMDD*' format, can be specified.

Example

```
SELECT UNIX_TIMESTAMP('1970-01-02'), UNIX_TIMESTAMP();

   unix_timestamp('1970-01-02')   unix_timestamp()
==================================================
```

```
                           540000         1270196737
```

UTC_DATE Function

Description

The UTC_DATE function returns the UTC date in 'YYYY-MM-DD' format.

Syntax

```
UTC_DATE()
```

Example

```
SELECT UTC_DATE();
  utc_date()
==============
  01/12/2011
```

UTC_TIME Function

Description

The UTC_TIME function returns the UTC time in 'HH:MM:SS' format.

Syntax

```
UTC_TIME()
```

Example

```
SELECT UTC_TIME();
  utc_time()
==============
  10:35:52 AM
```

WEEK Function

Description

The WEEK function returns the week in the range of 0 to 53 from the specified parameter. You can specify the **DATE**, **TIMESTAMP**, and **DATETIME** types as parameters, and an **INTEGER** type will be returned.

You can omit the second parameter, *mode* and must input a value in the range of 0 to 7. You can set that a week starts from Sunday or Monday and the range of the return value is from 0 to 53 or 1 to 53 with this value. If you omit the *mode*, the system parameter, **default_week_format** value will be used. The *mode* value means as follows:

mode	Start Day of the Week	Range	The First Week of the Year
0	Sunday	0~53	The first week that Sunday is included in the year
1	Monday	0~53	The first week that more than three days are included in the year
2	Sunday	1~53	The first week in the year that includes a Sunday
3	Monday	1~53	The first week in the year that includes more than three days
4	Sunday	0~53	The first week in the year that includes more than three days
5	Monday	0~53	The first week in the year that includes a Sunday
6	Sunday	1~53	The first week in the year that includes more than three days
7	Monday	1~53	The first week in the year that includes a Sunday

If the *mode* value is one of 0, 1, 4 or 5, and the date corresponds to the last week of the previous year, the WEEK function will return 0. The purpose is to see what nth of the year the week is so it returns 0 for the 52nd week of the year 1999.

```
SELECT YEAR('2000-01-01'), WEEK('2000-01-01',0);
   year('2000-01-01')   week('2000-01-01', 0)
=============================================
                2000                       0
```

To see what nth the week is based on the year including the start day of the week, use 0, 2, 5 or 7 as the *mode* value.

```
SELECT WEEK('2000-01-01',2);
    week('2000-01-01', 2)
========================
                      52
```

Syntax

```
WEEK(date[, mode])
```

- *date* : Date
- *mode* : Value in the range of 0 to 7

Example

```
SELECT WEEK('2010-04-05');
   week('2010-04-05', 0)
========================
                      14

SELECT WEEK('2010-04-05 12:34:56',2);
   week('2010-04-05 12:34:56',2)
===============================
                              14

SELECT WEEK('2010-04-05 12:34:56.7890',4);
   week('2010-04-05 12:34:56.7890',4)
====================================
                                  14

SELECT WEEK ('12:34:56');

ERROR: Conversion error in date format.

SELECT WEEK('2010-04-05',8);

ERROR: Conversion error in date format.
```

WEEKDAY Function

Description

The WEEKDAY function returns the day of week in the range of 0 to 6 (0: Sunday, 1: Monday, ..., 6: Saturday) from the specified parameter. The day of week index is same as the ODBC standards. You can specify **DATE**, **TIMESTAMP**, **DATETIME** types as parameters and an **INTEGER** type will be returned.

Syntax

```
WEEKDAY(date)
```

- *date* : Date

Example

```
SELECT WEEKDAY('2010-09-09');
   weekday('2010-09-09')
========================
                       3

SELECT WEEKDAY('2010-09-09 13:16:00');
```

```
   weekday('2010-09-09 13:16:00')
=================================
                                3

SELECT WEEKDAY('10:28:00');

ERROR: Conversion error in date format.
```

YEAR Function

Description

The YEAR function returns the year in the range of 1 to 9999 from the specified parameter. You can specify **DATE**, **TIMESTAMP**, and **DATETIME** types as parameters, and an **INTEGER** type will be returned.

Syntax

```
YEAR(date)
```

- *date* : Date

Example

```
SELECT YEAR('2010-10-04');
   year('2010-10-04')
=====================
                 2010

SELECT YEAR('2010-10-04 12:34:56');
   year('2010-10-04 12:34:56')
==============================
                          2010

SELECT YEAR('2010-10-04 12:34:56.7890');
   year('2010-10-04 12:34:56.7890')
===================================
                               2010

SELECT YEAR('12:34:56');

ERROR: Conversion error in date format.
```

Data Type Conversion Functions and Operators

CAST Operator

Description

The **CAST** operator can be used to explicitly cast one data type to another in the **SELECT** statement. A query list or a value expression in the **WHERE** clause can be cast to another data type.

Depending on the situation, data type can be automatically converted without suing the **CAST** operator. For more information, see Implicit Type Conversion.

See Converting the String of Date/Time Data Type into Data/Time Type regarding to convert the string of date/time type into date/time type.

The following table shows a summary of explicit type conversions (casts) using the **CAST** operator in CUBRID.

	EN	AN	VC	FC	VB	FB	BLOB	CLOB	D	T	UT	DT	S	MS	SQ
EN	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No
AN	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No
VC	Yes	Yes	Yes[1]	Yes*	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
FC	Yes	Yes	Yes*	Yes*	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No

VB	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No
FB	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No
BLOB	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No
CLOB	No	No	Yes	Yes	Yes	Yes	No	Yes	No	No	No	No	No	No	No
D	No	No	Yes	Yes	No	No	No	No	Yes	No	Yes	Yes	No	No	No
T	No	No	Yes	Yes	No	No	No	No	No	Yes	No	No	No	No	No
UT	No	No	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	No	No	No
DT	No	No	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	No	No	No
S	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
MS	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
SQ	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes

[1] In this case, the **CAST** operation is allowed only when the value expression and the data type to be cast have the same character code set.

Data Type Key

- **EN** : Exact numeric data type (**INTEGER**, **SMALLINT**, **BIGINT, NUMERIC**, **DECIMAL**)
- **AN** : Approximate numeric data type (**FLOAT/REAL**, **DOUBLE PRECISION**, **MONETARY**)
- **VC** : Variable-length character string (**VARCHAR**(*n*), **NCHAR VARYING**(*n*))
- **FC** : Fixed-length character string (**CHAR**(*n*), **NCHAR**(*n*))
- **VB**: Variable-length bit string (**BIT VARYING**(*n*))
- **FB** : Fixed-length bit string (**BIT**(*n*))
- **BLOB** : Binary data that is stored outside DB
- **CLOB** : String data that is stored inside DB
- **D** : Date (**DATE**)
- **T** : Time (**TIME**)
- **UT** : Timestamp (**TIMESTAMP**)
- **S** : Set (**SET**)
- **MS** : Multiset (**MULTISET**)
- **SQ** : Sequence set (**LIST**, **SEQUENCE**)

Syntax

```
CAST (cast_operand AS cast_target)

cast_operand :
• value expression
• NULL

cast_target :
• data type
```

- *cast_operand* : Declares the value to cast to a different data type.
- *cast_target* : Specifies the type to cast to.

Example

The following is an example of explicitly casting and returning a **VARCHAR** record in kg unit to a **FLOAT**.

```
--operation after casting character as INT type returns 2
SELECT (1+CAST ('1' AS INT));
  (1+ cast('1' as integer))
===========================
                          2
```

```
--cannot cast the string which is out of range as SMALLINT
SELECT (1+CAST('1234567890' AS SMALLINT));

ERROR: Cannot coerce value of domain "character" to domain "smallint".
--operation after casting returns 1+1234567890
SELECT (1+CAST('1234567890' AS INT));
 (1+ cast('1234567890' as integer))
====================================
                          1234567891

--'1234.567890' is casted to 1235 after rounding up
SELECT (1+CAST('1234.567890' AS INT));
 (1+ cast('1234.567890' as integer))
======================
  1236

--'1234.567890' is casted to string containing only first 5 letters.
SELECT (CAST('1234.567890' AS CHAR(5)));
 ( cast('1234.567890' as char(5)))
======================
  '1234.'

--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS CHAR(5)));

ERROR: Cannot coerce value of domain "numeric" to domain "character".
--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS CHAR(11)));
 ( cast(1234.567890 as char(11)))
======================
  '1234.567890'

--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS VARCHAR));
 ( cast(1234.567890 as varchar))
======================
  '1234.567890'

--string can be casted to time/date types only when its literal is correctly specified
SELECT (CAST('2008-12-25 10:30:20' AS TIMESTAMP));
 ( cast('2008-12-25 10:30:20' as timestamp))
=============================================
  10:30:20 AM 12/25/2008

SELECT (CAST('10:30:20' AS TIME));
 ( cast('10:30:20' as time))
==================================================
  10:30:20 AM

--string can be casted to TIME type when its literal is same as TIME’s.
SELECT (CAST('2008-12-25 10:30:20' AS TIME));
 ( cast('2008-12-25 10:30:20' as time))
========================================
  10:30:20 AM

--string can be casted to TIME type after specifying its type of the string
SELECT (CAST(TIMESTAMP'2008-12-25 10:30:20' AS TIME));
 ( cast(timestamp '2008-12-25 10:30:20' as time))
==================================================
  10:30:20 AM

SELECT CAST('abcde' AS BLOB);
 cast('abcde' as blob)
======================
file:/home1/user1/db/tdb/lob/ces_743/ces_temp.00001283232024309172_1342
```

Remark

- **CAST** is allowed only between data types having the same character set.
- If you cast an approximate data type to integer type, the number is rounded to zero decimal places.
- If you cast a numeric data type to string character type, it should be longer than the length of significant digits + decimal point. An error occurs otherwise.

- If you cast a character string type A to a character string type B, B should be longer than the A. The end of character string is truncated otherwise.
- If you cast a character string type A to a date-time date type B, it is converted only when literal of A and B type match one another. An error occurs otherwise.
- You must explicitly do type casting for numeric data stored in a character string so that an arithmetic operation can be performed.

DATE_FORMAT Function

Description

The **DATE_FORMAT** function converts the value of strings with **DATE** format ('*YYYY-MM-DD*' or '*MM/DD/YYYY*') or that of date/time data type (**DATE**, **TIMESTAMP**, **DATETIME**) to specified date/time format and then return the value with the **VARCHAR** data type.

Syntax

```
DATE_FORMAT(date, format)
```

- *date* : A value of strings with the **DATE** format ('*YYYY-MM-DD*' or '*MM/DD/YYYY*') or that of date/time data type (**DATE**, **TIMESTAMP**, **DATETIME**) can be specified .
- *format* : Specifies the output format. Use a string that contains '%' as a specifier. See the following table to specify the format. Date/Time formats described in the following Date/Time Format 2 table are used in **DATE_FORMAT** function, and TIME_FORMAT Function, and STR_TO_DATE Function.

Default Date/Time Format

Date/Time Type	Default Output Format
DATE	'MM/DD/YYYY'
TIME	'HH:MI:SS AM'
TIMESTAMP	'HH:MI:SS AM MM/DD/YYYY'
DATETIME	'HH:MI:SS.FF AM MM/DD/YYYY

Date/Time Format 2

format Value	Meaning
%a	Weekday, English abbreviation (Sun, …, Sat)
%b	Month, English abbreviation (Jan, …, Dec)
%c	Month (1, …, 12)
%D	Day of the month, English ordinal number (1st, 2nd, 3rd, ...)
%d	Day of the month, two-digit number (01, …, 31)
%e	Day of the month (1, …, 31)
%f	Microseconds, three-digit number (000, …, 999)
%H	Hour, 24-hour based, number with at least two--digit (00, …, 23, …, 100, …)
%h	Hour, 12-hour based two-digit number (01, …, 12)
%I	Hour, 12-hour based two-digit number (01, …, 12)
%i	Minutes, two-digit number (00, …, 59)
%j	Day of year, three-digit number (001, …, 366)
%k	Hour, 24-hour based, number with at least one-digit (0, …, 23, …, 100, …)
%l	Hour, 12-hour based (1, …, 12)
%M	Month, English string (January, …, December)
%m	Month, two-digit number (01, …, 12)

%p	AM or PM
%r	Time, 12-hour based, hour:minute:second (hh:mm:ss AM or hh:mm:ss PM)
%S	Seconds, two-digit number (00, …, 59)
%s	Seconds, two-digit number (00, …, 59)
%T	Time, 24-hour based, hour:minute:second (hh:mm:ss)
%U	Week, two-digit number, week number of the year with Sunday being the first day Week (00, …, 53)
%u	Week, two-digit number, week number of the year with Monday being the first day (00, …, 53)
%V	Week, two-digit number, week number of the year with Sunday being the first day Week (00, …, 53) (Available to use in combination with %X)
%v	Week, two-digit number, week number of the year with Monday being the first day (00, …, 53) (Available to use in combination with %X)
%W	Weekday, English string (Sunday, …, Saturday)
%w	Day of the week, number index (0=Sunday, …, 6=Saturday)
%X	Year, four-digit number calculated as the week number with Sunday being the first day of the week (0000, …, 9999) (Available to use in combination with %V)
%x	Year, four-digit number calculated as the week number with Monday being the first day of the week (0000, …, 9999) (Available to use in combination with %V)
%Y	Year, four-digit number (0001, …, 9999)
%y	Year, two-digit number (00, 01, …, 99)
%%	Output the special character "%" as a string
%x	Output an arbitrary character x as a string out of English letters that are not used as format specifiers.

Example

```
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%W %M %Y');
 date_format('2009-10-04 22:23:00', '%W %M %Y')
======================
  'Sunday October 2009'


SELECT DATE_FORMAT('2007-10-04 22:23:00', '%H:%i:%s');
 date_format('2007-10-04 22:23:00', '%H:%i:%s')
======================
  '22:23:00'

SELECT DATE_FORMAT('1900-10-04 22:23:00', '%D %y %a %d %m %b %j');
 date_format('1900-10-04 22:23:00', '%D %y %a %d %m %b %j')
======================
  '4th 00 Thu 04 10 Oct 277'


SELECT DATE_FORMAT('1999-01-01', '%X %V');
 date_format('1999-01-01', '%X %V')
======================
  '1998 52'
```

TIME_FORMAT Function

Description

The **TIME_FORMAT** function converts the value of strings with **TIME** format ('*HH-MI-SS)* or that of date/time data type (**DATE**, **TIMESTAMP**, **DATETIME**) to specified date/time format and then return the value with the **VARCHAR** data type.

Syntax

```
TIME_FORMAT(time, format)
```

- *time* : A value of string with **TIME** (*HH*:*MI*:*SS*) or that of date/time data type ((**DATE**, **TIMESTAMP**, **DATETIME**) an be specified.
- *format* : Specifies the output format. Use a string that contains '%' as a specifier. See the table of the Date/Time Format 2 table. If un-related format specifier is used, the English letters themselves are displayed.

Example

```
SELECT TIME_FORMAT('22:23:00', '%H %i %s');
 time_format('22:23:00', '%H %i %s')
======================
  '22 23 00'

SELECT TIME_FORMAT('23:59:00', '%H %h %i %s %f');
 time_format('23:59:00', '%H %h %i %s %f')
======================
  '23 11 59 00 000'

SELECT SYSTIME, TIME_FORMAT(SYSTIME, '%T');
 SYS_TIME     time_format( SYS_TIME , '%T')
===================================
  08:46:53 PM  '20:46:53'
```

TO_CHAR Function (date_time)

Description

The **TO_CHAR** function converts the value of strings with **TIME** format (*HH*:*MI*:*SS*) or that of date/time type **(TIME**, **TIMESTAMP**, **DATETIME**) by Date/Time Format 1 and then return the value with the **VARCHAR** data type. If a format argument is not specified, it converts the value based by default format. If a format which is not corresponding to the given value, an error is returned.

Syntax

```
TO_CHAR( date_time [, format [, date_lang_string_literal ]] )

date_time :
• date
• time
• timestamp
• datetime
• NULL

format :
• character strings (see Date/Time Format 1 )
• NULL

date_lang_string_literal : (see date_lang_string_literal)

• 'en_US'
• 'ko_KR'
```

- *date_time* : Specifies an expression that returns date-time type string. If the value is **NULL**, **NULL** is returned.
- *format* : Specifies a format of return value. If a format is not specified, the default format is used. If the value is **NULL**, **NULL** is returned.

- *date_lang_string_literal* : Specifies a language applied to a return value (refer to date_lang_string_literal). The default value is 'en_US'. You can modify the value by specifying the **CUBRID_DATE_LANG** environment variable.

Default Date/Time Format

Date/Time Type	Default Output Format
DATE	'MM/DD/YYYY'
TIME	'HH:MI:SS AM'
TIMESTAMP	'HH:MI:SS AM MM/DD/YYYY'
DATETIME	'HH:MI:SS.FF AM MM/DD/YYYY'

Date/Time Format 1

Format Element	Description
CC	Century
YYYY, **YY**	Year with 4 numbers, Year with 2 numbers
Q	Quarter (1, 2, 3, 4; January - March = 1)
MM	Month (01-12; January = 01) Note : MI represents the minute of hour.
MONTH	Month in characters
MON	Abbreviated month name
DD	Day (1 - 31)
DAY	Day of the week in characters
DY	Abbreviated day of the week
D or **d**	Day of the week in numbers (1 - 7)
AM or **PM**	AM/PM
A.M. or **P.M.**	AM/PM with periods
HH or **HH12**	Hour (1 -12)
HH24	Hour (0 - 23)
MI	Minute (0 - 59)
SS	Second (0 - 59)
FF	Millsecond (0-999)
- / , . ; : "text"	Punctuation and quotation marks are represented as they are in the result

Example of date_lang_string_literal

Format Element	Date_lang_string_literal	
	'en_US'	**'ko_KR'**
MONTH	JANUARY	1 월
MON	JAN	1
DAY	MONDAY	월요일
DY	MON	월
Month	January	1 월
Mon	Jan	1

Day	Monday	월요일
Dy	Mon	월
month	january	1 월
mon	jan	1
day	monday	월요일
Dy	mon	월
AM	AM	오전
Am	Am	오전
am	am	오전
A.M.	A.M.	오전
A.m.	A.m.	오전
a.m.	a.m.	오전
PM	AM	오전
Pm	Am	오전
pm	am	오전
P.M.	A.M.	오전
P.m.	A.m.	오전
p.m.	a.m	오전

The Number of Digits Format

Format Element	**Number of Digits**
MONTH(Month, month)	9 (ko_KR : 4)
MON(Mon, mon)	3 (ko_KR : 2)
DAY(Day, day)	9 (ko_KR : 6)
DY(Dy, dy)	3 (ko_KR : 2)
HH12, HH24	2
"text"	The length of the text
Other formats	Same as the length of the format

Example

```
--creating a table having date/time type columns
CREATE TABLE datetime_tbl(a TIME, b DATE, c TIMESTAMP, d DATETIME);
INSERT INTO datetime_tbl VALUES(SYSTIME, SYSDATE, SYSTIMESTAMP, SYSDATETIME);

--selecting a VARCHAR type string from the data in the specified format
SELECT TO_CHAR(b, 'DD, DY , MON, YYYY') FROM datetime_tbl;
 to_char(b, 'DD, DY , MON, YYYY', 'en_US')
======================
  '04, THU , FEB, 2010'
```

```
SELECT TO_CHAR(c, 'HH24:MI, DD, MONTH, YYYY') FROM datetime_tbl;
 to_char(c, 'HH24:MI, DD, MONTH, YYYY', 'en_US')
======================
  '16:50, 04, FEBRUARY , 2010'

SELECT TO_CHAR(c, 'HH24:MI:FF, DD, MONTH, YYYY') FROM datetime_tbl;

ERROR: Invalid format.

SELECT TO_CHAR(d, 'HH12:MI:SS:FF pm, YYYY-MM-DD-DAY') FROM datetime_tbl;
 to_char(d, 'HH12:MI:SS:FF pm, YYYY-MM-DD-DAY', 'en_US')
======================
  '04:50:11:624 pm, 2010-02-04-THURSDAY '

SELECT TO_CHAR(TIMESTAMP'2009-10-04 22:23:00', 'Day Month yyyy');
 to_char(timestamp '2009-10-04 22:23:00', 'Day Month yyyy', 'en_US')
======================
  'Sunday October 2009'
```

TO_CHAR Function (number)

Description

The **TO_CHAR** function converts a Number Format or numeric data type to a character string according to the number format and returns it. The type of the return value is **VARCHAR**. If the number format has not been specified as an argument, all significant digits are converted to a character string according to the default format.

Syntax

```
TO_CHAR(number_argument[, format_argument ])

number_argument :
• numeric(decimal)
• integer
• smallint
• bigint
• float(real)
• double
• NULL

format_argument :
• character strings (see Number Format)
• NULL
```

- *number_argument* : Specifies an expression that returns numeric data type string. If the input value is **NULL**, **NULL** is returned. If the input value is character type, the character itself is returned.
- *format_argument* : Specifies a format of return value. If format is not specified, all significant digits are returned as character string by default. If the value is **NULL**, **NULL** is returned.

Number Format

Format Element	Example	Description
9	9999	The number of 9's represents the number of significant digits to be returned. If the number of significant digits specified in the format is not sufficient, only the decimal part is rounded. If it is less than the number of digits in an integer, # is outputted. If the number of significant digits specified in the format is sufficient, the part preceding the integer part is filled with space characters and the decimal part is filled with 0.
0	0999	If the number of significant digits specified in the format is sufficient, the part preceding the integer part is filled with 0, not space characers before the value is returned.
S	S9999	Outputs the negative/positive sign in the specified position. These signs can be used only at the beginning of character string.

C	C9999	Returns the ISO currency code at the specified position.
,(comma)	9,999	Returns a comma (",") at the specified position. Multiple commas are allowed in the format.
.(percimal point)	9.999	Outputs the decimal point (".") that distinguishes the integer and the decimal part at a specified position. Only one decimal point is allowed in the format.
EEEE	9.99EEEE	Returns a scientific notation number.

Example

```
--selecting a string casted from a number in the specified format
SELECT TO_CHAR(12345,'S999999'), TO_CHAR(12345,'S099999');

============================================
  ' +12345'             '+012345'


SELECT TO_CHAR(1234567,'C9,999,999,999');
======================
  '    $1,234,567'

SELECT TO_CHAR(123.4567,'99'), TO_CHAR(123.4567,'999.99999'),
TO_CHAR(123.4567,'99999.999');
 to_char(123.4567, '99', 'en_US')    to_char(123.4567, '999.99999',
'en_US')    to_char(123.4567, '99999.999', 'en_US')
==========================================================
  '##'                  '123.45670'           '  123.457'


SELECT TO_CHAR(1.234567,'99.999EEEE'), TO_CHAR(1.234567E-4);
 to_char(1.234567, '99.999EEEE', 'en_US')    to_char(1.234567E-4)
============================================
  '1.235E+00'           '0.0001234567'
```

TO_DATE Function

Description

The **TO_DATE** function interprets a character string based on the date format given as an argument, converts it to a **DATE** type value, and returns it. For the format, see TO_CHAR Function (date_time). If a format is not specified, the "MM/DD/YYYY" format is applied by default.

Syntax

```
TO_DATE(string_argument[,format_argument[,date_lang_string_literal]])

string_argument :
• character strings
• NULL

format_argument :
• character strings (see Date/Time Format 1)
• NULL

date_lang_string_literal : (see date lang string literal)
• 'en_US'
• 'ko_KR'
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **DATE** type. See the "Default Date-Time Format" table of TO_CHAR Function (date_time). If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the input value to be applied. You can modify the value by using the **CUBRID_DATE_LANG** environment.

Example

```
--selecting a date type value casted from a string in the specified format

SELECT TO_DATE('12/25/2008');
 to_date('12/25/2008')
===============================================
  12/25/2008

SELECT TO_DATE('25/12/2008', 'DD/MM/YYYY');
 to_date('25/12/2008', 'DD/MM/YYYY', 'en_US')
===============================================
  12/25/2008

SELECT TO_DATE('081225', 'YYMMDD');
 to_date('081225', 'YYMMDD', 'en_US')
===============================================
  12/25/2008

SELECT TO_DATE('2008-12-25', 'YYYY-MM-DD');
 to_date('2008-12-25', 'YYYY-MM-DD', 'en_US')
===============================================
  12/25/2008
```

TO_DATETIME Function

Description

The **TO_DATETIME** function interprets a character string based on the date-time format given as an argument, converts it to a **DATETIME** type value, and returns it. For the format, see TO_CHAR Function (date_time). If format is not specified, the "HH:MI:SS.FF [am|pm] MM/DD/YYYY" format is applied by default.

Syntax

```
TO_DATETIME(string_argument[,format_argument[,date_lang_string_literal]])

string_argument :
• character strings
• NULL

format_argument :
• character strings (see the table Date/Time Format 1)
• NULL

date_lang_string_literal : (see the table Example of date_lang_string_literal)
• 'en_US'
• 'ko_KR'
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **DATETIME** type. See the "Default Date-Time Format" table of TO_CHAR Function (date_time). If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the input value to be applied. You can modify the value by using the **CUBRID_DATE_LANG** environment.

Example

```
--selecting a datetime type value casted from a string in the specified format

SELECT TO_DATETIME('13:10:30 12/25/2008');
 to_datetime('13:10:30 12/25/2008')
=====================================
  01:10:30.000 PM 12/25/2008

SELECT TO_DATETIME('08-Dec-25 13:10:30.999', 'YY-Mon-DD HH24:MI:SS.FF');
 to_datetime('08-Dec-25 13:10:30.999', 'YY-Mon-DD HH24:MI:SS.FF', 'en_US')
=====================================
  01:10:30.999 PM 12/25/2008

SELECT TO_DATETIME('DATE: 12-25-2008 TIME: 13:10:30.999', '"DATE:" MM-DD-YYYY "TIME:"
HH24:MI:SS.FF');
```

```
 to_datetime('DATE: 12-25-2008 TIME: 13:10:30.999', '"DATE:" MM-DD-YYYY "TIME:"
HH24:MI:SS.FF', 'en_US')
=====================================
  01:10:30.999 PM 12/25/2008
```

TO_NUMBER Function

Description

The **TO_NUMBER** function interprets a character string based on the number format given as an argument, converts it to a **NUMERIC** type value, and returns it. If the number format is not specified, returns all significant digits that are included in the character string as **NUMERIC** type numbers by default.

Syntax

```
TO_NUMBER(string_argument[, format_argument ])

string_argument :
• character strings
• NULL

format_argument :
• character strings
• NULL
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **NUMBER** type. See the "Number Format" table of TO_CHAR Function (number). If the value is **NULL**, an error is returned.

Example

```
--selecting a number casted from a string in the specified format
SELECT TO_NUMBER('-1234');
 to_number('-1234')
============================================
  -1234


SELECT TO_NUMBER('12345','999999');
 to_number('12345', '999999')
============================================
  12345


SELECT TO_NUMBER('$12,345.67','C99,999.999');
 to_number('$12,345.67', 'C99,999.999')
======================
  12345.670


SELECT TO_NUMBER('12345.67','99999.999');
 to_number('12345.67', '99999.999')
============================================
  12345.670
```

TO_TIME Function

Description

The **TO_TIME** function interprets a character string based on the time format given as an argument, converts it to a **TIME** type value, and returns it. For the format, see TO_CHAR Function (date_time). If a format is not specified, the "HH:MI:SS" format is applied by default.

Syntax

```
TO_TIME(string_argument[,format_argument [,date_lang_string_literal]]):

string_argument :
```

```
• character strings
• NULL

format_argument :
• character strings (refer to Date/Time Format 1)
• NULL

date_lang_string_literal : (refer to date_lang_string_literal)
• 'en_US'
• 'ko_KR'
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **TIME** type. See the "Default Date-Time Format" table of TO_CHAR Function (date_time). If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the input value to be applied. You can modify the value by using the **CUBRID_DATE_LANG** environment.

Example

```
--selecting a time type value casted from a string in the specified format

SELECT TO_TIME ('13:10:30');
 to_time('13:10:30')=============================================
  01:10:30 PM

SELECT TO_TIME('HOUR: 13 MINUTE: 10 SECOND: 30', '"HOUR:" HH24 "MINUTE:" MI "SECOND:" SS');
 to_time('HOUR: 13 MINUTE: 10 SECOND: 30', '"HOUR:" HH24 "MINUTE:" MI "SECOND:" SS',
'en_US')=============================================
  01:10:30 PM

SELECT TO_TIME ('13:10:30', 'HH24:MI:SS');
 to_time('13:10:30', 'HH24:MI:SS', 'en_US')
=============================================
  01:10:30 PM

SELECT TO_TIME ('13:10:30', 'HH12:MI:SS');

ERROR: Conversion error in date format.
```

TO_TIMESTAMP Function

Description

The **TO_TIMESTAMP** function interprets a character string based on the time format given as an argument, converts it to a **TIMESTAMP** type value, and returns it. For the format, see TO_CHAR Function (date_time). If a format is not specified, the "HH:MI[:SS] [am|pm] MM/DD/YYYY" format is applied by default.

Syntax

```
TO_TIMESTAMP(string_argument[, format_argument[,date_lang_string_literal]])

string_argument :
• character strings
• NULL

format_argument :
• character strings (refer to Date/Time Format 1 table)
• NULL

date_lang_string_literal : (refer to date_lang_string_literal table)
• 'en_US'
• 'ko_KR'
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **TIMESTAMP** type. See the "Default Date-Time Format" table of TO_CHAR Function (date_time). If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the input value to be applied. You can modify the value by using the **CUBRID_DATE_LANG** environment.

Example

```
--selecting a timestamp type value casted from a string in the specified format

SELECT TO_TIMESTAMP('13:10:30 12/25/2008');
 to_timestamp('13:10:30 12/25/2008')
======================================
  01:10:30 PM 12/25/2008

SELECT TO_TIMESTAMP('08-Dec-25 13:10:30', 'YY-Mon-DD HH24:MI:SS');
 to_timestamp('08-Dec-25 13:10:30', 'YY-Mon-DD HH24:MI:SS', 'en_US')
======================================
  01:10:30 PM 12/25/2008

SELECT TO_TIMESTAMP('YEAR: 2008 DATE: 12-25 TIME: 13:10:30', '"YEAR:" YYYY "DATE:" MM-DD
"TIME:" HH24:MI:SS');
 to_timestamp('YEAR: 2008 DATE: 12-25 TIME: 13:10:30', '"YEAR:" YYYY "DATE:" MM-DD "TIME:"
HH24:MI:SS', 'en_US')
======================================
  01:10:30 PM 12/25/2008
```

Aggregate Functions

AVG Function

Description

The **AVG** function calculates the arithmetic average of the value of an expression representing all rows. Only one *expression* is specified as a parameter. You can get the average without duplicates by using the **DISTINCT** or **UNIQUE** keyword in front of the expression or the average of all values by omitting the keyword or by using **ALL**.

Syntax

```
AVG ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression* : Specifies an expression that returns a numeric value. A collection expression cannot be specified.
- **ALL** : Calculates an average value for all data (default).
- **DISTINCT** or **UNIQUE** : Calculates an average value without duplicates.

Example

The following is an example that returns the average number of gold medals Korea won in Olympics. (demodb)

```
SELECT AVG(gold)
FROM participant
WHERE nation_code = 'KOR';
```

Result value : 9

COUNT Function

Description

The **COUNT** function returns the number of of rows returned by a query. If an asterisk (*) is specified, the number of all rows satisfying the condition (including the rows with the **NULL** value) is returned. If the **DISTINCT** or **UNIQUE** keyword is specified in front of the expression, only the number of rows that have a unique value (excluding the rows with the **NULL** value) is returned after duplicates have been removed. Therefore, the value returned is always an integer and **NULL** is never returned.

A column that has collection type and object domain (user-defined class or multimedia class) can also be specified in the *expression*.

Syntax

```
COUNT ( * | [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression* : Specifies an expression.
- **ALL** : Gets the number of rows given in the *expression* (default).
- **DISTINCT** or **UNIQUE** : Gets the number of rows without duplicates.

Example

The following is an example that returns the number of Olympic Games that had a mascot. (demodb)

```
SELECT COUNT(*)
FROM olympic
WHERE mascot IS NOT NULL;
```

Result value : 9

GROUP_CONCAT Function

Description

The **GROUP_CONCAT** function connects the values that are not **NULL** in the group and returns the character string in the **VARCHAR** type. If there are no rows of query result or there are only **NULL** values, **NULL** will be returned.

The maximum size of the return value follows the configuration of the system parameter, **group_concat_max_len**. The default is **1024** bytes, the minimum value is 4 bytes and the maximum value is 33,554,432 bytes. If it exceeds the maximum value, **NULL** will be returned.

To remove the duplicate values, use the **DISTINCT** clause. The default separator for the group result values is comma (,). To represent the separator explicitly, add the character string to use as a separator in the **SEPARATOR** clause and after that. If you want to remove separators, enter empty strings after the **SEPARATOR** clause.

If the non-character string type is passed to the result character string, an error will be returned.

To use the **GROUP_CONCAT** function, you must meet the following conditions.

- Only one expression (or a column) is allowed for an input parameter.
- Sorting with **ORDER BY** is available only in the the expression used as a parameter.
- The character string used as a separator allows not only character string type but also allows other types.

Syntax

```
GROUP_CONCAT([DISTINCT] {col | expression}
             [ORDER BY {col | unsigned_int} [ASC | DESC]]
             [SEPARATOR str_val])
```

- *expression* : Operation returning numerical values or character strings
- *str_val* : Character string to use as a separator
- **DISTINCT** : Removes duplicate values from the result.
- **ORDER BY** : Specifies the order of result values.
- **SEPARATOR** : Specifies the separator to divide the result values. If you omit it, the default character, comma (,) will be used as a separator.

Example

```
SELECT GROUP_CONCAT(s_name) FROM code;
  group_concat(s_name)
======================
  'X,W,M,B,S,G'

SELECT GROUP_CONCAT(s_name ORDER BY s_name SEPARATOR ':') from code;
  group_concat(s_name order by s_name separator ':')
======================
  'B:G:M:S:W:X'

CREATE TABLE t(i int);
INSERT INTO t VALUES (4),(2),(3),(6),(1),(5);

SELECT GROUP_CONCAT(i*2+1 ORDER BY 1 SEPARATOR '') FROM t;
```

```
  group_concat(i*2+1 order by 1 separator '')
======================
  '35791113'
```

MAX Function

Description

The **MAX** function gets the greatest value of expressions of all rows. Only one *expression* is specified.

For expressions that return character strings, the string that appears later in alphabetical order becomes the maximum value; for those that return numbers, the greatest value becomes the maximum value.

Syntax

```
MAX ( [ { { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression* : Specifies an expression that returns a numeric or string value. A collection expression cannot be specified.
- **ALL** : Gets the maximum value for all data (default).
- **DISTINCT** or **UNIQUE** : Gets the maximum value without duplicates.

Example

The following is an example that returns the maximum number of gold medals Korea won in the Olympics. (demodb)

```
SELECT MAX(gold) FROM participant WHERE nation_code = 'KOR';
    max(gold)
=============
           12
```

MIN Function

Description

The **MIN** function gets the smallest value of expressions of all rows. Only one *expression* is specified.

For expressions that return character strings, the string that appears earlier in alphabetical order becomes the minimum value; for those that return numbers, the smallest value becomes the minimum value.

Syntax

```
MIN ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression* : Specifies an expression that returns a numeric or string value. A collection expression cannot be specified.
- **ALL** : Gets the minimum value for all data (default).
- **DISTINCT** or **UNIQUE** : Gets the maximum value without duplicates.

Example

The following is an example that returns the minimum number of gold medals Korea won in the Olympics. (demodb)

```
SELECT MIN(gold) FROM participant WHERE nation_code = 'KOR';
    min(gold)
=============
            7
```

STDDEV, STDDEV_POP Function

Description

The **STDDEV** function returns a standard deviation of the expression values of all rows. Only one *expression* is specified as a parameter. You can get the standard deviation without duplicates by inserting the **DISTINCT** or

UNIQUE keyword in front of the expression, or get the standard deviation of all values by omitting the keyword or by using **ALL**.

The return value may be different from the actual evaluation value because it follows the type of the expression specified as a parameter.

Syntaxs

```
STDDEV( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL] expression )
```

- *expression* : Specifies an expression that returns a numeric value.
- **ALL** : Calculates the standard deviation for all data (default).
- **DISTINCT** or **UNIQUE** : Calculates the standard deviation without duplicates.

Example

The following is an example that returns the standard deviation of gold medals Korea won in the Olympics. (demodb)

```
SELECT host_year, gold FROM participant WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

    host_year          gold
==========================
         2004             9
         2000             8
         1996             7
         1992            12
         1988            12

SELECT STDDEV(gold), STDDEV(CAST (gold AS FLOAT)) FROM participant
WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

  stddev(gold)   stddev( cast(gold as float))
============================================
             2                 2.302172e+000
```

STDDEV_SAMP Function

Description

The **STDDEV_SAMP** function calculates the sample standard deviation. Only one *expression* is specified as a parameter. If the **DISTINCT** or **UNIQUE** keyword is included, it calculates the sample standard deviation after deleting the duplicates; if the keyword is omitted or is **ALL**, it calculates the sample standard deviation for all values.

The return value is the same as the square root of the VAR_SAMP Function return value and it is a **DOUBLE** type. If there are no rows that can be used for calculating a result, **NULL** will be returned.

The following are the formulas applied to the function.

$\mathrm{STDDEV\_SAMP} = [\ \{\ 1\ /\ (N-1)\ \}\ *\ \mathrm{SUM}(\ \{\ x_I - \mathrm{mean}(x)\ \}^2\ )\ ]^{1/2}$

- SUM : Sum
- mean : Mean

Syntax

```
STDDEV_SAMP( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL] expression )
```

- *expression* : Specifies one operation that returns a numerical value.
- **ALL** : Is used to calculate the standard deviation for all values. It is the default value.
- **DISTINCT** or **UNIQUE** : Is used to calculate the standard deviation for the unique values without duplicates.

Example

```
CREATE TABLE test_table (d DOUBLE);
INSERT INTO test_table VALUES(78), (63.65), (230.54), (32), (17.2), (195.7689), (57.57);

SELECT STDDEV_SAMP(d) FROM test_table;
            stddev_samp(d)
==========================
    8.287199825135663e+01

SELECT STDDEV_SAMP(POWER(d,2)+d*2+1) FROM test_table;
  stddev_samp( power(d, 2)+d*2+1)
=================================
           2.155888498702931e+04

TRUNCATE TABLE test_table;
SELECT STDDEV_SAMP(d) FROM test_table;
                stddev_samp(d)
==========================
                      NULL
```

SUM Function

Description

The **SUM** function returns the sum of expressions of all rows. Only one *expression* is specified as a parameter. You can get the sum without duplicates by inserting the **DISTINCT** or **UNIQUE** keyword in front of the expression, or get the sum of all values by omitting the keyword or by using **ALL**.

Syntax

```
SUM ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

You can specify a single-value expression as a input for **SUM** function.

- *expression* : Specifies an expression that returns a numeric value.
- **ALL** : Gets the sum for all data (default).
- **DISTINCT** or **UNIQUE** : Gets the sum of unique values without duplicates

Example

The following is an example that outputs the top 10 countries and the total number of gold medals based on the sum of gold medals won in the Olympics. (demodb)

```
SELECT nation_code, SUM(gold) FROM participant GROUP BY nation_code
ORDER BY SUM(gold) DESC
FOR ORDERBY_NUM() BETWEEN 1 AND 10 ;

=== <Result of SELECT Command in Line 1> ===

  nation_code             sum(gold)
===================================
  'USA'                         190
  'CHN'                          97
  'RUS'                          85
  'GER'                          79
  'URS'                          55
  'FRA'                          53
  'AUS'                          52
  'ITA'                          48
  'KOR'                          48
  'EUN'                          45


10 rows selected.
```

VARIANCE Function

Description

The **VARIANCE** function returns a variance of expression values of all rows. Only one *expression* is specified as a parameter. You can get the variance without duplicates by using the **DISTINCT** or **UNIQUE** keyword in front of the expression or the variance of all values by omitting the keyword or by using **ALL**.

The return value may be different from the actual evaluation value because it follows the type of the expression specified as a parameter.

The following is a formula that is applied to the function.

$$\frac{\left[\mathrm{SUM}(x^2) - \frac{(\mathrm{SUM}(x))^2}{n}\right]}{(n-1)}$$

Syntax

```
VARIANCE( [DISTINCT | UNIQUE | ALL] expression )
```

- *expression* : Specifies an expression that returns a numeric value.
- **ALL** : Gets the variance for all values (default).
- **DISTINCT** or **UNIQUE** : Gets the variance of unique values without duplicates.

Example

The following is an example that returns the variance of the number of gold medals Korea has won from 1988 to 2004 in the Olympic Games. (demodb)

```
SELECT VARIANCE(gold), VARIANCE(CAST (gold AS FLOAT)) FROM participant
WHERE nation_code = 'KOR';
=== <Result of SELECT Command in Line 1> ===

  variance(gold)  variance( cast(gold as float))
================================================
               5                   5.299995e+000
```

VAR_SAMP Function

Description

The **VAR_SAMP** function returns the sample variance. The denominator is the number of all rows - 1. Only one *expression* is specified as a parameter. If the **DISTINCT** or **UNIQUE** keyword is included, it calculates the sample variance after deleting the duplicates, and if the keyword is omitted or is **ALL**, it calculates the sample variance for all values.

The return value is a **DOUBLE** type. If there are no rows that can be used for calculating a result, **NULL** will be returned.

The following are the formulas applied to the function.

VAR_POP = { 1 / (N-1) } * SUM({ x_I - AVG(x) }^{2})

- SUM : Sum
- AVG : Average

```
VAR_SAMP( [ DISTINCT | UNIQUE | ALL] expression )
```

- *expression* : Specifies one expression to return the numeric.
- **ALL** : Is used to calculate the sample variance of unique values without duplicates. It is the default value.
- **DISTINCT** or **UNIQUE** : Is used to calculate the sample variance for the unique values without duplicates.

Example

```
CREATE TABLE test_table (d double);
INSERT INTO test_table VALUES(78), (63.65), (230.54), (32), (17.2), (195.7689), (57.57);
SELECT VAR_SAMP(d) FROM test_table;
               var_samp(d)
==========================
    6.867768094172856e+03

SELECT VAR_SAMP(POWER(d,2)+d*2+1) FROM test_table;
  var_samp( power(d, 2)+d*2+1)
==============================
        4.647855218839577e+08

TRUNCATE TABLE test_table;
SELECT VAR_SAMP(d) FROM test_table;
                var_samp(d)
==========================
                      NULL
```

Click Counter Functions

INCR, DECR Function

Description

The **INCR** function increments the column's value given as a parameter for a SELECT statement by 1. The **DECR** function decrements the value of the column by 1.

Syntax

```
SELECT [ qualifier ] select_expression
[ { TO | INTO } variable [ {, variable }...; ] ]
...;
select_expression :
 *
 table_name. *
 [expression | counter_expression] [ {, expression |
counter_expression}...]

counter_expression :
INCR(path_expression)
```

The **INCR** and **DECR** functions are called "click counters" and can be effectively used to increase the number of post views for a Bulletin Board System (BBS) type of web service. In a scenario where you want to SELECT a post and immediately increase the number of views by 1 using an **UPDATE** statement, you can view the post and increment the number at the same time by using the **INCR** function in a single **SELECT** statement.

The **INCR** function increments the column value specified as an argument. Only integer type numbers can be used as arguments. If the value is **NULL**, the **INCR** function returns the **NULL**. That is, a value must be valid in order to be incremented by the **INCR** function. The **DECR** function decrements the column value specified as a parameter.

If an **INCR** function is specified in the SELECT statement, the **COUNTER** value is incremented by 1 and the query result is displayed with the values before the increment. Furthermore, the **INCR** function does not increment the value of the tuple affected by the query process but rather the one affected by the final result.

Remark

- The **INCR/DECR** function executes independent of user-defined transactions and is applied automatically to the database by the top operation internally used in the system, apart from the transaction's **COMMIT/ROLLBACK**.
- When multiple **INCR/DECR** functions are specified in a single SELECT statement, the failure of any of the **INCR/DECR** functions leads to the failure of all of them.
- The **INCR/DECR** functions apply only to top-level SELECT statements. **SUB SELECT** statements such as **INSERT** ... **SELECT** ... statement and **UPDATE** table **SET** col = **SELECT** ... statement are not supported. The following is an example where the **INCR** function is not allowed.

```
SELECT b.content, INCR(b.read_count) FROM (SELECT * FROM board WHERE id = 1) AS b
```

- If the **SELECT** statement with **INCR/DECR** function(s) returns more than one row as a result, it is treated as an error. The final result must have only one row to be considered valid.
- The **INCR/DECR** function can be used only in numerical domains. Applicable domains are limited to integer data types such as **SMALLINT** and **INTEGER**. They cannot be used in other domains.
- When the **INCR** function is called, the value to be returned will be the current value, while the value to be stored will be the current value + 1. Execute the following statement to select the value to be stored as the result :

```
SELECT content, INCR(read_count) + 1 FROM board WHERE id = 1;
```

- If the defined maximum value of the domain is exceeded, the **INCR** function initializes the column value to 0. Likewise, the column value is also initialized to 0 when the **DECR** function applies to the minimum value.
- Data inconsistency can occur because the **INCR/DECR** functions are executed regardless of **UPDATE** trigger. The example below shows the database inconsistency in that situation.

```
CREATE TRIGGER event_tr BEFORE UPDATE ON event EXECUTE REJECT;
SELECT INCR(players) FROM event WHERE gender='M';
```

Example

Suppose that the following three rows of data were inserted into the 'board' table.

```
CREATE TABLE board (
id  INT, title  VARCHAR(100), content  VARCHAR(4000), read_count  INT );
INSERT INTO board VALUES (1, 'aaa', 'text...', 0);
INSERT INTO board VALUES (2, 'bbb', 'text...', 0);
INSERT INTO board VALUES (3, 'ccc', 'text...', 0);
```

The following is an example of incrementing the value of the 'read_count' column in a data whose 'id' value is 1 using the **INCR** function.

```
SELECT content, INCR(read_count) FROM board WHERE id = 1;
  content                read_count
===================================
  'text...'                       0
```

In the example, the column value becomes read_count + 1 as a result of the **INCR** function in the **SELECT** statement. You can check the result using the following **SELECT** statement.

```
SELECT content, read_count FROM board WHERE id = 1;
  content                read_count
===================================
  'text...'                       1
```

ROWNUM Functions

ROWNUM/INST_NUM() Function

Description

The **ROWNUM** function returns the number representing the order of the records that will be generated by the query result. The first result record is assigned 1, and the second result record is assigned 2.

ROWNUM and **INST_NUM()** can be used in the **SELECT** statement, and **GROUPBY_NUM()** can be used in the **SELECT** statement with **GROUP BY** clauses. The **ROWNUM** function can be used to limit the number of result records of the query in several ways. For example, it can be used to search only the first 10 records or to return even or odd number records.

The **ROWNUM** function has a result value as an integer, and can be used wherever an expression is valid such as the **SELECT** or **WHERE** clause. However, it is not allowed to compare the result of the **ROWNUM** function with the attribute or the correlated subquery.

Syntax

```
INST_NUM()
ROWNUM
```

Remark

- The **ROWNUM** function specified in the **WHERE** clause works the same as the **INST_NUM()** function. Whereas **INST_NUM()** is a scalar function, **GROUPBY_NUM()** is a kind of an aggregate function. In a **SELECT** statement with a **GROUP BY** clause, **GROUPBY_NUM()** must be used instead of **INST_NUM()**.
- The **ROWNUM** function belongs to each **SELECT** statement. That is, if a **ROWNUM** function is used in a subquery, it returns the sequence of the subquery result while it is being executed. Internally, the result of the **ROWNUM** function is generated right before the searched record is written to the query result set. At this moment, the counter value that generates the serial number of the result set records increases.
- If an **ORDER BY** clause is included in the **SELECT** statement, the value of the **ROWNUM** function specified in the **WHERE** clause is generated before sorting for the **ORDER BY** clause. If a **GROUP BY** clause is included in the **SELECT** statement, the value of the **GROUPBY_NUM()** function specified in the **HAVING** clause is calculated after the query results are grouped. After the sorting process is completed using the **ORDER BY** clause, you need to use the **ORDERBY_NUM()** function in the **ORDER BY** clause in order to get a sequence of the result records.
- The **ROWNUM** function can also be used in SQL statements such as **INSERT**, **DELETE** and **UPDATE** in addition to the **SELECT** statement. For example, as in the query **INSERT INTO** *table_name* **SELECT** ... **FROM** ... **WHERE** ..., you can search for part of the row from one table and then insert it into another by using the **ROWNUM** function in the **WHERE** clause.

Example

The following is an example (demodb) that returns country names ranked first to fourth based on the number of gold medals in the 1988 Olympics.

```
--Limiting 4 rows using ROWNUM in the WHERE condition
SELECT  * FROM
(SELECT nation_code FROM participant WHERE host_year = 1988
     ORDER BY gold DESC) AS T
WHERE ROWNUM <5;
  nation_code
======================
  'URS'
  'GDR'
  'USA'
  'KOR'

--Limiting 4 rows using FOR ORDERBY_NUM()
SELECT ROWNUM, nation_code FROM participant WHERE host_year = 1988
ORDER BY gold DESC
FOR ORDERBY_NUM() < 5;
       rownum  nation_code
===================================
          156  'URS'
          155  'GDR'
          154  'USA'
          153  'KOR'

--Unexpected results : ROWNUM operated before ORDER BY
SELECT ROWNUM, nation_code FROM participant
WHERE host_year = 1988 AND ROWNUM < 5
ORDER BY gold DESC;
       rownum  nation_code
===================================
            1  'ZIM'
            2  'ZAM'
            3  'ZAI'
            4  'YMD'
```

GROUPBY_NUM() Function

Description

The **GROUPBY_NUM()** function is used with the **ROWNUM()** or **INST_NUM()** function to limit the number of result rows. The difference is that the **GROUPBY_NUM()** function is combined after the **GROUP BY … HAVING** clause to give order to a result that has been already sorted. In addition, while the **INST_NUM()** function is a scalar function, the **GROUPBY_NUM()** function is kind of an aggregate function.

That is, when retrieving only some of the result rows by using **ROWNUM** in a condition clause of the **SELECT** statement that includes the **GROUP BY** clause, **ROWNUM** is applied first and then group sorting by **GROUP BY** is performed. On the other hand, when retrieving only some of the result rows by using the **GROUPBY_NUM()** function, **ROWNUM** is applied to the result of group sorting by **GROUP BY**.

Syntax

```
GROUPBY_NUM()
```

Example

The following is an example that searches for the fastest record in the previous five Olympic Games from the history table. (demodb)

```
--Group-ordering first and then limiting rows using GROUPBY_NUM()
SELECT host_year, MIN(score) FROM history
GROUP BY host_year HAVING GROUPBY_NUM() BETWEEN 1 AND 5;
    host_year  min(score)
===================================
         1968  '8.9'
         1980  '01:53.0'
         1984  '13:06.0'
         1988  '01:58.0'
         1992  '02:07.0'

--Limiting rows first and then Group-ordering using ROWNUM
SELECT host_year, MIN(score) FROM history
WHERE ROWNUM BETWEEN 1 AND 5 GROUP BY host_year;
    host_year  min(score)
===================================
         2000  '03:41.0'
         2004  '01:45.0'
```

ORDERBY_NUM() Function

Description

The **ORDERBY_NUM()** function is used with the **ROWNUM()** or **INST_NUM()** function to limit the number of result rows. The difference is that the **ORDERBY_NUM()** function is combined after the ORDER BY clause to give order to a result that has been already sorted.

That is, when retrieving only some of the result rows by using **ROWNUM** in a condition clause of the **SELECT** statement that includes the **ORDER BY** clause, **ROWNUM** is applied first and then group sorting by **ORDER BY** is performed. On the other hand, when retrieving only some of the result rows by using the **ORDER_NUM()** function, **ROWNUM** is applied to the result of sorting by **ORDER BY**.

Syntax

```
FOR ORDERBY_NUM()
```

Example

The following is an example of searching athlete names ranked 3rd to 5th and their records in the history table. (demodb)

```
--Ordering first and then limiting rows using FOR ORDERBY_NUM()
SELECT athlete, score FROM history
ORDER BY score FOR ORDERBY_NUM() BETWEEN 3 AND 5;
  athlete               score
============================================
  'Luo Xuejuan'         '01:07.0'
  'Rodal Vebjorn'       '01:43.0'
  'Thorpe Ian'          '01:45.0'

--Limiting rows first and then Ordering using ROWNUM
SELECT athlete, score FROM history
WHERE ROWNUM BETWEEN 3 AND 5 ORDER BY score;
  athlete               score
```

```
============================================
  'Thorpe Ian'          '01:45.0'
  'Thorpe Ian'          '03:41.0'
  'Hackett Grant'       '14:43.0'
```

Information Functions

CURRENT_USER, USER

Description

CURRENT_USER and **USER** are used interchangeably. They return the user name that is currently logged in to the database as a string.

USER() and **SYSTEM_USER()** are used interchangeably. They return the user name with a host name.

Syntax

```
CURRENT_USER
USER
USER()
SYSTEM_USER()
```

Example

```
--selecting the current user on the session
SELECT USER;
   CURRENT_USER
======================
  'PUBLIC'

SELECT USER(), CURRENT_USER;
   user()                CURRENT_USER
============================================
  'PUBLIC@cdbs006.cub'  'PUBLIC'

--selecting all users of the current database from the system table
SELECT name, id, password FROM db_user;
  name                           id  password
=========================================================
  'DBA'                        NULL  NULL
  'PUBLIC'                     NULL  NULL
  'SELECT_ONLY_USER'           NULL  db_password
  'ALMOST_DBA_USER'            NULL  db_password
  'SELECT_ONLY_USER2'          NULL  NULL
```

DATABASE(), SCHEMA()

Description

DATEBASE() returns the name of the currently-connected database as a **VARCHAR** type string. **DATABASE** and **SCHEMA** are used interchangeably.

Syntax

```
DATABASE()
SCHEMA()
```

Example

```
SELECT DATABASE(), SCHEMA();
;xr
   database()            schema()
============================================
  'demodb'              'demodb'
```

DEFAULT Function

Description

The **DEFAULT** function returns a default value defined for a column. If a default value has not been specified for the given column, **NULL** or an error is returned. If any of constraints is not defined or the **UNIQUE** constraint is defined for the column where a default value is not defined, **NULL** is returned. If **NOT NULL** or **PRIMARY KEY** constraint is defined, an error is returned.

Syntax

```
DEFAULT(column_name)
```

Example

```
CREATE TABLE info_tbl(id INT DEFAULT 0, name VARCHAR)
INSERT INTO info_tbl VALUES (1,'a'),(2,'b'),(NULL,'c');

3 rows affected.

SELECT id, DEFAULT(id) FROM info_tbl;
           id   default(id)
=============================
            1             0
            2             0
         NULL             0
```

INDEX_CARDINALITY Function

Description

The **INDEX_CARDINALITY** function returns the index cardinality in a table. The index cardinality is the number of unique values defining the index. The index cardinality can be applied even to the partial key of the multiple column index and displays the number of the unique value for the partial key by specifying the column location with the third parameter.

The return value is 0 or a positive integer and if any of the input parameters is **NULL**, **NULL** will be returned. If tables or indexes that are input parameters are not found, or *key_pos* is out of range, **NULL** will be returned.

For the table and the index names which are the first and the second input parameters, they cannot be passed as **NCHAR** or **VARNCHAR** types.

Syntax

```
INDEX_CARDINALITY(table, index, key_pos)
```

- *table* : Table name
- *index* : Index name that exists in the *table*
- *key_pos* : Partial key location *key_pos* starts from 0 and has a range that is smaller than the number of columns that composes the key. That is, the *key_pos* of the first column is 0. For the single column index, it is 0. It can be one of the following types.
- Character string that can be converted to a numeric type. **NCHAR** and **VARNCHAR** are not supported.
- Numeric type that can be converted to an integer type. The **FLOAT** or the **DOUBLE** types will be the value converted by the **ROUND** function.

Example

```
CREATE TABLE t1( i1 INTEGER ,
i2 INTEGER not null,
i3 INTEGER unique,
s1 VARCHAR(10),
s2 VARCHAR(10),
s3 VARCHAR(10) UNIQUE);

CREATE INDEX i_t1_i1 ON t1(i1 DESC);
```

```
CREATE INDEX i_t1_s1 ON t1(s1(7));
CREATE INDEX i_t1_i1_s1 on t1(i1,s1);
CREATE UNIQUE INDEX i_t1_i2_s2 ON t1(i2,s2);

INSERT INTO t1 VALUES (1,1,1,'abc','abc','abc');
INSERT INTO t1 VALUES (2,2,2,'zabc','zabc','zabc');
INSERT INTO t1 VALUES (2,3,3,'+abc','+abc','+abc');

SELECT INDEX_CARDINALITY('t1','i_t1_i1_s1',0);
   index_cardinality('t1', 'i_t1_i1_s1', 0)
===========================================
                                          2

SELECT INDEX_CARDINALITY('t1','i_t1_i1_s1',1);
   index_cardinality('t1', 'i_t1_i1_s1', 1)
===========================================
                                          3

SELECT INDEX_CARDINALITY('t1','i_t1_i1_s1',2);
   index_cardinality('t1', 'i_t1_i1_s1', 2)
===========================================
                                       NULL

SELECT INDEX_CARDINALITY('t123','i_t1_i1_s1',1);
  index_cardinality('t123', 'i_t1_i1_s1', 1)
=============================================
                                         NULL
```

LAST_INSERT_ID Function

Description

The **LAST_INSERT_ID** function returns the value created at the end of the **AUTO_INCREMENT** column of all tables.

If no values are **INSERT**ed successfully, the last successful value will be maintained, and the SQL statement on execution does not affect the **LAST_INSERT_ID**() value. If you enter multiple rows with one **INSERT** statement, the **LAST_INSERT_ID**() will return the input row value entered at the end. If the execution result of the previous SQL statement returns an error, the **LAST_INSERT_ID**() value is not defined, and the rollback can not recover the **LAST_INSERT_ID**() value as the previous transaction value.

You can not check the **LAST_INSERT_ID**() value used in the trigger, outside trigger.

The created ID is maintained independently for the connection of each client.

Syntax

```
LAST_INSERT_ID()
```

Example

```
CREATE TABLE ss (id INT AUTO_INCREMENT NOT NULL PRIMARY KEY, text VARCHAR(32));
INSERT into ss VALUES(NULL,’cubrid’);
SELECT LAST_INSERT_ID();

     last_insert_id()
=======================
                     1

INSERT INTO ss VALUES(NULL,’database’),(NULL,’manager’);
SELECT LAST_INSERT_ID();

     last_insert_id()
=======================
                     3
```

LIST_DBS Function

Description

The **LIST_DBS** function outputs the list of all databases in the CUBRID database server, separated by blanks.

Syntax

```
LIST_DBS()
```

Example

```
SELECT LIST_DBS();
  dbs
======================
  'testdb demodb'
```

ROW_COUNT Function

Description

The **ROW_COUNT** function returns the number of rows updated (**UPDATE**, **INSERT**, **DELETE)** by the previous statement. Note that the **ROW_COUNT** function execution area at the SQL level is limited to the client session in which the SQL was created. If this function is called after executing SQL with the **;run** or **;xrun** command, it returns -1.

Syntax

```
ROW_COUNT()
```

Example

```
SELECT * FROM info_tbl;

=== <Result of SELECT Command in Line 1> ===

           id  name
===================================
            1  'a'
            2  'b'
         NULL  'c'

INSERT INTO info_tbl VALUES (4,'d'),(5, 'e');
SELECT ROW_COUNT();
;xr

=== <Result of SELECT Command in Line 2> ===

   row_count()
===============
              2

DELETE FROM info_tbl WHERE id IN (4,5);
SELECT ROW_COUNT();
;xr

=== <Result of SELECT Command in Line 2> ===

   row_count()
===============
              2


SELECT ROW_COUNT();

=== <Result of SELECT Command in Line 1> ===

   row_count()
===============
             -1
```

CURRENT_USER, USER

Description

CURRENT_USER and **USER** are used interchangeably. They return the user name that is currently logged in to the database as a string.

USER() and **SYSTEM_USER()** are used interchangeably. They return the user name with a host name.

Syntax

```
CURRENT_USER
USER
USER()
SYSTEM_USER()
```

Example

```
--selecting the current user on the session
SELECT USER;
   CURRENT_USER
======================
  'PUBLIC'

SELECT USER(), CURRENT_USER;
   user()                CURRENT_USER
============================================
  'PUBLIC@cdbs006.cub'  'PUBLIC'

--selecting all users of the current database from the system table
SELECT name, id, password FROM db_user;
  name                           id  password
=========================================================
  'DBA'                        NULL  NULL
  'PUBLIC'                     NULL  NULL
  'SELECT_ONLY_USER'           NULL  db_password
  'ALMOST_DBA_USER'            NULL  db_password
  'SELECT_ONLY_USER2'          NULL  NULL
```

VERSION Function

Description

Returns the version character string representing the CUBRID server version.

Syntax

```
VERSION()
```

Example

```
SELECT VERSION();
   version()
=====================
  '8.3.1.2015'
```

Encryption Function

MD5 Function

Description

Returns the MD5 128-bit checksum for the input character string. The result value is displayed as a character string that is expressed in 32 hexadecimals, which you can use to create hash keys, for example.

The return value is a **VARCHAR**(32) type and if an input parameter is **NULL**, **NULL** will be returned.

Syntax

```
MD5(string)
```

- *string* : Input character string. If a value that is not a **VARCHAR** type is entered, it will be converted to **VARCHAR**.

Example

```
SELECT MD5('cubrid');
   md5('cubrid')
======================
  '685c62385ce717a04f909047d0a55a16'

SELECT MD5(255);
   md5(255)
======================
  'fe131d7f5a6b38b23cc967316c13dae2'
SELECT MD5('01/01/2010');

   md5('01/01/2010')
======================
  '4a2f373c30426a1b8e9cf002ef0d4a58'

SELECT MD5(CAST('2010-01-01' as DATE));
   md5( cast('2010-01-01' as date))
======================
  '4a2f373c30426a1b8e9cf002ef0d4a58'
```

Conditional Operators and Functions

CASE

Description

The **CASE** expression uses the SQL statement to perform an **IF** ... **THEN** statement. When a result of comparison expression specified in a **WHEN** clause is true, a value specified in **THEN** value is returned. A value specified in an **ELSE** clause is returned otherwise. If no **ELSE** clause exists, **NULL** is returned.

Syntax

```
CASE control_expression simple_when_list
[ else_clause ]
END
CASE searched_when_list
[ else_clause ]
END

simple_when :
WHEN expression THEN result

searched_when :
WHEN search_condition THEN result

else_clause :
ELSE result

result :
expression | NULL
```

The CASE expression must end with the END keyword. A *control_expression* argument and an *expression argument* in *simple_when* expression should be comparable data types. The data types of *result* specified in the **THEN** ... **ELSE** statement should all same, or they can be convertible to common data type.

The data type for a value returned by the **CASE** expression is determined based on the following rules.

- If data types for result specified in the **THEN** statement are all same, a value with the data type is returned.

- If data types can be convertible to common data type even though they are not all same, a value with the data type is returned.
- If any of values for *result* is a variable length string, a value data type is a variable length string. If values for *result* are all a fixed length string, the longest character string or bit string is returned.
- If any of values for result is an approximate numeric data type, a value with a numeric data type is returned. The number of digits after the decimal point is determined to display all significant digits.

Example

```
--creating a table
CREATE TABLE case_tbl( a INT);
INSERT INTO case_tbl VALUES (1);
INSERT INTO case_tbl VALUES (2);
INSERT INTO case_tbl VALUES (3);
INSERT INTO case_tbl VALUES (NULL);

--case operation with a search when clause
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
FROM case_tbl;
            a  case when a=1 then 'one' when a=2 then 'two' else 'other' end
===================================
            1  'one'
            2  'two'
            3  'other'
         NULL  'other'

--case operation with a simple when clause
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
FROM case_tbl;
            a  case a when 1 then 'one' when 2 then 'two' else 'other' end
===================================
            1  'one'
            2  'two'
            3  'other'
         NULL  'other'


--result types are converted to a single type containing all of significant figures
SELECT a,
       CASE WHEN a=1 THEN 1
            WHEN a=2 THEN 1.2345
            ELSE 1.234567890
       END
FROM case_tbl;
            a  case when a=1 then 1 when a=2 then 1.2345 else 1.234567890 end
===================================
            1  1.000000000
            2  1.234500000
            3  1.234567890
         NULL  1.234567890

--an error occurs when result types are not convertible
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 1.2345
       END
FROM case_tbl;
ERROR: Cannot coerce 'one' to type double.
```

COALESCE Function

Description

The **COALESCE** function has more than one expression as an argument. If a first argument is non-**NULL**, it is returned if it is **NULL**, a second argument is returned. If all expressions which have an argument are **NULL**, **NULL** is returned. Therefore, this function is generally used to replace NULL with other default value. All expressions with an argument must be identical or each data type must be convertible to one another.

Syntax

```
COALESCE(expression [, ...])

result :
expression | NULL
```

COALESCE(*a*, *b*) works the same as the **CASE** statement as follows:

```
CASE WHEN a IS NOT NULL
THEN a
ELSE b
END
```

Example

```
SELECT * FROM case_tbl;
            a
=============
            1
            2
            3
         NULL

--substituting a default value 10.0000 for NULL valuse
SELECT a, COALESCE(a, 10.0000) FROM case_tbl;
            a  coalesce(a, 10.0000)
===================================
            1  1.0000
            2  2.0000
            3  3.0000
         NULL  10.0000
```

DECODE Function

Description

As well as a **CASE** expression, the **DECODE** function performs the same functionality as the **IF** ... **THEN** ... **ELSE** statement. It compares the *expression* argument with *search* argument, and returns the *result* corresponding to *search* that has the same value. It returns *default* if there is no *search* with the same value, and returns **NULL** if *default* is omitted. An expression argument and a search argument to be comparable should be same or convertible each other. The number of digits after the decimal point is determined to display all significant digits including valid number of all *result*.

Syntax

```
DECODE( expression, search, result [, search, result]* [, default] )

result :
result | default | NULL
```

DECODE(*a*, *b*, *c*, *d*, *e*) works the same as the **CASE** statement as follows:

```
CASE WHEN a = b THEN c
WHEN a= c THEN d
ELSE e
END
```

Example

```
SELECT * FROM case_tbl;
            a
=============
            1
            2
            3
         NULL

--Using DECODE function to compare expression and search values one by one
SELECT a, DECODE(a, 1, 'one', 2, 'two', 'other') FROM case_tbl;
            a  decode(a, 1, 'one', 2, 'two', 'other')
===================================
            1  'one'
            2  'two'
            3  'other'
         NULL  'other'


--result types are converted to a single type containing all of significant figures
SELECT a, DECODE(a, 1, 1, 2, 1.2345, 1.234567890) FROM case_tbl;
            a  decode(a, 1, 1, 2, 1.2345, 1.234567890)
===================================
            1  1.000000000
            2  1.234500000
            3  1.234567890
         NULL  1.234567890

--an error occurs when result types are not convertible
SELECT a, DECODE(a, 1, 'one', 2, 'two', 1.2345) FROM case_tbl;

ERROR: Cannot coerce 'one' to type double.
```

IF Function

Description

The **IF** function returns *expression2* if the value of the arithmetic expression specified as the first parameter is **TRUE**, or *expression3* if the value is **FALSE** or **NULL**. *expression2* and *expression3* which are returned as the result must be the same or of a convertible common type. If one is explicitly **NULL**, the result of the function follows the type of the non-**NULL** parameter.

Syntax

```
IF( expression1, expression2, expression3 )

result :
exrpession2 | expression3
```

IF(*a*, *b*, *c*) works the same as the **CASE** statement as follows:

```
CASE WHEN a IS TRUE THEN b
ELSE c
END
```

Example

```
SELECT * FROM case_tbl;
            a
=============
            1
            2
            3
         NULL
--IF function returns the second expression when the fist is TRUE
SELECT a, IF(a=1, 'one', 'other') FROM case_tbl;
            a   if(a=1, 'one', 'other')
===================================
            1  'one'
            2  'other'
```

```
            3  'other'
         NULL  'other'

--If function in WHERE clause
SELECT * FROM case_tbl WHERE IF(a=1, 1, 2) = 1;
            a
=============
            1
```

IFNULL, NVL Function

Description

The **IFNULL** function is working like the **NVL** function; however, only the **NVL** function supports set data type as well. The **IFNULL** function (which has two arguments) returns *expr1* if the value of the first expression is not **NULL** or returns *expr2,* otherwise. The data type of the result is determined as the type which can be converted from both *expr1* and *expr2* types; see the table below.

expr1 Type	expr2 Type	Type of NVL Return Value	Type of IFNULL Return Value
?	?	Error	VARCHAR
?	X	X type	VARCHAR
CHAR CHAR VARCHAR	VARCHAR NCHAR NCHAR	VARCHAR Error VARCHAR	VARCHAR VARCHAR VARCHAR
String type String type Number type	Number type Date/Time type Date/Time type	Error Error Error	VARCHAR VARCHAR VARCHAR
Date/Time type Number type	Date/Time type Number type	Common convertible type Common convertible type	VARCHAR Common convertible type
Collection type Collection type	Collection type Others	Common convertible type Error	Error Error

Syntax

```
IFNULL( expr1, expr2 )

result :
expr1 | expr2
```

IFNULL(*a*, *b*, *c*) works the same as the **CASE** statement as follows:

```
CASE WHEN a IS NULL THEN b
ELSE a
END
```

Example

```
SELECT * FROM case_tbl;
            a
=============
            1
            2
            3
         NULL

--returning a specific value when a is NULL
SELECT a, NVL(a, 10.0000) FROM case_tbl;
            a  nvl(a, 10.0000)
===================================
            1  1.0000
            2  2.0000
            3  3.0000
         NULL  10.0000
```

```
--IFNULL can be used instead of NVL and return values are converted to the string type
SELECT a, IFNULL(a, 'UNKNOWN') FROM case_tbl;
            a   ifnull(a, 'UNKNOWN')
===================================
            1  '1'
            2  '2'
            3  '3'
         NULL  'UNKNOWN'
```

NULLIF Function

Description

The **NULLIF** function returns **NULL** if the two expressions specified as the parameters are identical, and returns the first parameter value otherwise.

Syntax

```
NULLIF(expr1, expr2)

result :
expr1 | NULL
```

NULLIF(*a*, *b*) is the same of the **CASE** statement.

```
CASE
WHEN a = b THEN NULL
ELSE a
END
```

Example

```
SELECT * FROM case_tbl;
            a
=============
            1
            2
            3
         NULL

--returning NULL value when a is 1
SELECT a, NULLIF(a, 1) FROM case_tbl;
            a  nullif(a, 1)
===========================
            1          NULL
            2             2
            3             3
         NULL          NULL

--returning NULL value when arguments are same
SELECT NULLIF (1, 1.000)  FROM db_root;
  nullif(1, 1.000)
======================
  NULL

--returning the first value when arguments are not same
SELECT NULLIF ('A', 'a')  FROM db_root;
  nullif('A', 'a')
======================
  'A'
```

NVL2 Function

Description

Three parameters are specified for the **NVL** function. The function returns the second expression (*expr2*) if the first expression (*expr1*) is not **NULL**, and the third expression (*expr2*) if it is **NULL**.

Syntax

```
NVL2( expr1, expr2, expr3 )

result :
expr2 | expr3
```

Example

```
SELECT * FROM case_tbl;
            a
=============
            1
            2
            3
         NULL

--returning a specific value of INT type
SELECT a, NVL2(a, a+1, 10.5678) FROM case_tbl;
            a  nvl2(a, a+1, 10.5678)
====================================
            1                      2
            2                      3
            3                      4
         NULL                     11
```

Conditional Expressions

Basic Conditional Expressions

A conditional expression is an expression that is included in the **WHERE** clause of the **SELECT**, **UPDATE** and **DELETE** statements, and in the **HAVING** clause of the **SELECT** statement. There are simple comparison, **ANY**/**SOME**/**ALL**, **BETWEEN**, **EXISTS**, **IN**/**NOT IN**, **LIKE** and **IS NULL** conditional expressions, depending on the kinds of the operators combined.

A simple comparison conditional expression compares two comparable data values. Expressions or subqueries are specified as operands, and the conditional expression always returns **NULL** if one of the operands is **NULL**. The following table shows operators that can be used in the simple comparison conditional expressions. For details, see Comparison Operator.

Operators for Conditional Expressions

Comparison Operator	Description	Conditional Expression	Return Value
=	A value of left operand is the same as that of right operand.	1=2	0
<>, !=	A value of left operand is not the same as that of right operand.	1<>2	1
>	A value of left operand is greater than that of right operand.	1>2	0
<	A value of left operand is less than that of right operand.	1<2	1
>=	A value of left operand is equal to or greater than that of right operand.	1>=2	0
<=	A value of left operand is equal to or less than that of right operand.	1<=2	1

ANY/SOME/aLL Conditional Expressions

Description

Group conditional expressions that include quantifiers such as **ANY/SOME/aLL** perform comparison operation on one data value and on some or all values included in the list. A conditional expression that includes **ANY** or **SOME** returns **TRUE** if the value of the data on the left satisfies simple comparison with at least one of the values in the list specified as an operand on the right. A group conditional expression that includes **ALL** returns **TRUE** if the value of the data on the left satisfies simple comparison with all values in the list on the right.

When a comparison operation is performed on **NULL** in a group conditional expression that includes **ANY** or **SOME**, **UNKNOWN** or **TRUE** is returned as the result; when a comparison operation is performed on **NULL** in a group conditional expression that includes **ALL**, **UNKNOWN** or **FALSE** is returned.

Syntax

```
expression comp_op SOME expression
expression comp_op ANY expression
expression comp_op ALL expression
```

- *comp_op* : A comparison operator >, = or <= can be used.
- *expression* (left) : A single-value column, path expression, constant value or arithmetic function that produces a single value can be used.
- *expression* (right) : A column name, path expression, list (set) of constant values or subquery can be used. A list is a set represented within braces ({}). If a subquery is used, *expression* (left) and comparison operation on all results of the subquery execution is performed.

Example

```
--creating a table

CREATE TABLE condition_tbl (id int primary key, name char(10), dept_name VARCHAR, salary
INT);
INSERT INTO condition_tbl VALUES(1, 'Kim', 'devel', 4000000);
INSERT INTO condition_tbl VALUES(2, 'Moy', 'sales', 3000000);
INSERT INTO condition_tbl VALUES(3, 'Jones', 'sales', 5400000);
INSERT INTO condition_tbl VALUES(4, 'Smith', 'devel', 5500000);
INSERT INTO condition_tbl VALUES(5, 'Kim', 'account', 3800000);
INSERT INTO condition_tbl VALUES(6, 'Smith', 'devel', 2400000);
INSERT INTO condition_tbl VALUES(7, 'Brown', 'account', NULL);

--selecting rows where department is sales or devel
SELECT * FROM condition_tbl WHERE dept_name = ANY{'devel','sales'};
           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            6  'Smith     '          'devel'                   2400000

--selecting rows comparing NULL value in the ALL group conditions
SELECT * FROM condition_tbl WHERE salary > ALL{3000000, 4000000, NULL};
There are no results.

--selecting rows comparing NULL value in the ANY group conditions
SELECT * FROM condition_tbl WHERE salary > ANY{3000000, 4000000, NULL};
           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            5  'Kim       '          'account'                 3800000

--selecting rows where salary*0.9 is less than those salary in devel department
SELECT * FROM condition_tbl WHERE (
(0.9 * salary) < ALL (SELECT salary FROM condition_tbl
WHERE dept_name = 'devel')
```

```
);
           id  name                  dept_name                  salary
======================================================================
            6  'Smith     '          'devel'                   2400000
```

BETWEEN Conditional Expression

Description

The **BETWEEN** conditional expression makes a comparison to determine whether the data value on the left exists between two data values specified on the right. It returns **TRUE** even when the data value on the left is the same as a boundary value of the comparison target range. If **NOT** comes before the **BETWEEN** keyword, the result of a **NOT** operation on the result of the **BETWEEN** operation is returned.

i **BETWEEN** *g* **AND** *m* and the compound condition *i* **>= g AND** *i* <= *m* have the same effect.

Syntax

```
expression [ NOT ] BETWEEN expression AND expression
```

- *expression* : A column name, path expression, constant value, arithmetic expression or aggregate function can be used. For a character string expression, the conditions are evaluated in alphabetical order. If **NULL** is specified for at least one of the expressions, the **BETWEEN** predicate returns **UNKNOWN** as the result.

Example

```
--selecting rows where 3000000 <= salary <= 4000000
SELECT * FROM condition_tbl WHERE salary BETWEEN 3000000 AND 4000000;

SELECT * FROM condition_tbl WHERE (salary >= 3000000) AND (salary <= 4000000);
           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            5  'Kim       '          'account'                 3800000

--selecting rows where salary < 3000000 or salary > 4000000
SELECT * FROM condition_tbl WHERE salary NOT BETWEEN 3000000 AND 4000000;
           id  name                  dept_name                  salary
======================================================================
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            6  'Smith     '          'devel'                   2400000

--selecting rows where name starts from A to E
SELECT * FROM condition_tbl WHERE name BETWEEN 'A' AND 'E';
           id  name                  dept_name                  salary
======================================================================
            7  'Brown     '          'account'                    NULL
```

EXISTS Conditional Expression

Description

The **EXISTS** conditional expression returns **TRUE** if one or more results of the execution of the subquery specified on the right exist, and returns **FALSE** if the result of the operation is an empty set.

Syntax

```
EXISTS expression
```

- *expression* : Specifies a subquery and compares to determine whether the result of the subquery execution exists. If the subquery does not produce any result, the result of the conditional expression is **FALSE**.

Example

```
--selecting rows using EXISTS and subquery
SELECT 'raise' FROM db_root WHERE EXISTS(
```

```
SELECT * FROM condition_tbl WHERE salary < 2500000);
  'raise'
======================
  'raise'

--selecting rows using NOT EXISTS and subquery
SELECT 'raise' FROM db_root WHERE NOT EXISTS(
SELECT * FROM condition_tbl WHERE salary < 2500000);
There are no results.
```

IN Conditional Expression

Description

The **IN** conditional expression compares to determine whether the single data value on the left is included in the list specified on the right. That is, the predicate returns **TRUE** if the single data value on the left is an element of the expression specified on the right. If **NOT** comes before the **IN** keyword, the result of a **NOT** operation on the result of the **IN** operation is returned.

Syntax

```
expression [ NOT ] IN expression
```

- *expression* (left) : A single-value column, path expression, constant value or arithmetic function that produces a single value can be used.
- *expression* (right) : A column name, path *expression*, list (set) of constant values or subquery can be used. A list is a set represented within parentheses (()) or braces ({}). If a subquery is used, comparison with expression(left) is performed for all results of the subquery execution.

Example

```
--selecting rows where department is sales or devel
SELECT * FROM condition_tbl WHERE dept_name IN {'devel','sales'};

SELECT * FROM condition_tbl WHERE dept_name = ANY{'devel','sales'};
           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            6  'Smith     '          'devel'                   2400000

--selecting rows where department is neither sales nor devel
SELECT * FROM condition_tbl WHERE dept_name NOT IN {'devel','sales'};
           id  name                  dept_name                  salary
======================================================================
            5  'Kim       '          'account'                 3800000
            7  'Brown     '          'account'                    NULL
```

IS NULL Conditional Expression

Description

The **IS NULL** conditional expression compares to determine whether the expression specified on the left is **NULL**, and if it is **NULL**, returns **TRUE** and it can be used in the conditional expression. If **NOT** comes before the **NULL** keyword, the result of a **NOT** operation on the result of the **IS NULL** operation is returned.

Syntax

```
expression IS [ NOT ] NULL
```

- *expression* : A single-value column, path expression, constant value or arithmetic function that produces a single value can be used.

Example

```
SELECT * FROM condition_tbl WHERE salary IS NULL;
           id  name                  dept_name                  salary
======================================================================
            7  'Brown     '          'account'                    NULL

--selecting rows where salary is NOT NULL
SELECT * FROM condition_tbl WHERE salary IS NOT NULL;
           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            5  'Kim       '          'account'                 3800000
            6  'Smith     '          'devel'                   2400000

--simple conparison operation returns NULL when operand is NULL
SELECT * FROM condition_tbl WHERE salary = NULL;
There are no results.
```

ISNULL Function

Description

The **ISNULL** function performs a comparison to determine if the result of the expression specified as an argument is **NULL**. The function returns 1 if it is **NULL** or 0 otherwise. You can check if a certain value is **NULL**. This function is working like the **ISNULL** expression.

Syntax

```
ISNULL(expression)
```

- *expression* : An arithmetic function that has a single-value column, path expression, constant value is specified.

Example

```
--Using ISNULL function to select rows with NULL value
SELECT * FROM condition_tbl WHERE ISNULL(salary);
           id  name                  dept_name                  salary
======================================================================
            7  'Brown     '          'account'                    NULL
```

LIKE Conditional Expression

Description

The **LIKE** conditional expression compares patterns between character string data, and returns **TRUE** if a character string whose pattern matches the search word is found. Pattern comparison target domains are **CHAR**, **VARCHAR** and **STRING**. The **LIKE** search cannot be performed on an **NCHAR** or **BIT** type. If **NOT** comes before the **LIKE** keyword, the result of a **NOT** operation on the result of the **LIKE** operation is returned.

A wild card string corresponding to any character or character string can be included in the search word on the right of the **LIKE** operator. % (percent) and _ (underscore) can be used. .% corresponds to any character string whose length is 0 or greater, and _ corresponds to one character. An escape character is a character that is used to search for a wild card character itself, and can be specified by the user as another character (**NULL**, alphabet, or number_ whose length is 1. See below for an example of using a character string that includes wild card or escape characters.

Syntax

```
expression [ NOT ] LIKE expression [ ESCAPE char]
```

- *expression* (left) : Specify the data type column of the character string. Pattern comparison, which is case-sensitive, starts from the first character of the column.

- *expression* (right) : Enter the search word. A character string with a length of 0 or greater is required. Wild card characters (% or _) can be included as the pattern of the search word. The length of the character string is 0 or greater.
- **ESCAPE** *char* : **NULL**, alphabet, or number is allowed for *char*. If the string pattern of the search word includes "_" or "%" itself, an ESCAPE character must be specified. For example, if you want to search for the character string "10%" after specifying backslash (\) as the ESCAPE character, you must specify "10\%" for the expression (right). If you want to search for the character string "C:\", you can specify "C:\\" for the expression (right).

Remark

LIKE search may not work properly for data entered in multi-byte character set environment such as utf-8. This is because byte units for string comparison operation differ depending on the character sets. You can get normal results by adding a parameter(**single_byte_compare**=yes) to the **cubrid.conf** file that enables string comparison in a single-byte units, and restarting the DB.

For details about character sets supported in CUBRID, see Definition and Characteristics. For details about the single_byte_compare parameter, see Other Parameters.

Whether to detect the escape characters of the LIKE conditional expression is determined depending on the configuration of **no_backslash_escapes** and **require_like_escape_character** in the **cubrid.conf** file. For more information, see Statement/Type-Related Parameters.

Example

```
--selection rows where name contains lower case 's', not upper case
SELECT * FROM condition_tbl WHERE name LIKE '%s%';
           id  name                  dept_name                     salary
======================================================================
            3  'Jones     '          'sales'                      5400000

--selection rows where second letter is 'O' or 'o'
SELECT * FROM condition_tbl WHERE UPPER(name) LIKE '_O%';
           id  name                  dept_name                     salary
======================================================================
            2  'Moy       '          'sales'                      3000000
            3  'Jones     '          'sales'                      5400000

--selection rows where name is 3 characters

SELECT * FROM condition_tbl WHERE name LIKE '___';
           id  name                  dept_name                     salary
======================================================================
            1  'Kim       '          'devel'                      4000000
            2  'Moy       '          'sales'                      3000000
            5  'Kim       '          'account'                    3800000
```

Data Manipluation

SELECT

Overview

Description

The **SELECT** statement specifies columns that you want to retrieve from a table.

Syntax

```
SELECT [ <qualifier> ] <select_expressions>
    [ { TO | INTO } <variable_comma_list> ]
    [ FROM <extended_table_specification_comma_list> ]
    [ WHERE <search_condition> ]
    [ GROUP BY {col_name | expr} [ ASC | DESC ],...[ WITH ROLLUP ] ]
    [ HAVING  <search_condition> ]
    [ ORDER BY {col_name | expr} [ ASC | DESC ],... [ FOR <orderby_for_condition> ] ]
    [ LIMIT [offset,] row_count ]
    [ USING INDEX { index_name [,index_name,...] | NONE }]

<qualifier> ::= ALL | DISTINCT | DISTINCTROW | UNIQUE

<select_expressions> ::= * | <expression_comma_list> | *, <expression_comma_list>

<extended_table_specification_comma_list> ::=
<table specification> [ {, <table specification> | <join table specification> }... ]

<table_specification> ::=
 <single_table_spec> [ <correlation> ] [ WITH (lock_hint) ]|
 <metaclass_specification> [ <correlation> ] |
 <subquery> <correlation> |
 TABLE ( <expression> ) <correlation>

<correlation> ::= [ AS ] <identifier> [ ( <identifier_comma_list> ) ]

<single_table_spec> ::= [ ONLY ] <table_name> |
                      ALL <table_name> [ EXCEPT <table_name> ]

<metaclass_specification> ::= CLASS <class_name>

<join_table_specification> ::=
[ INNER | [ LEFT | RIGHT [ OUTER ] ] JOIN <table specification> ON <search condition>

lock_hint :
READ UNCOMMITTED

<orderby_for_condition> ::=
ORDERBY_NUM() { BETWEEN int AND int } |
    { { = | =< | < | > | >= } int } |
    IN ( int, ...)
```

- *qualifier* : A qualifier. It can be omitted. When omitted, it is set to **ALL**.
- **ALL** : Retrieves all records of the table.
- **DISTINCT** : Retrieves only records with unique values without allowing duplicates. **DISTINCT** and **DISTINCTROW** are used interchangeably.
- **UNIQUE** : Like **DISTINCT**, retrieves only records with unique values without allowing duplicates.
- *select_expression* :
- * : By using **SELECT** * statement, you can retrieve all the columns from the table specified in the **FROM** clause.
- *expression_comma_list* : *expression* can be a path expression, variable or table name. All general expressions including arithmetic operations can also be used. Use a comma (,) to separate each expression in the list.
  You can specify aliases by using the **AS** keyword for columns or expressions to be queried. Specified aliases are used as column names in **GROUP BY**, **HAVING**, **ORDER BY** and **FOR** clauses. The position index of a column

is assigned based on the order in which the column was specified. The starting value is 1.
As **AVG**, **COUNT**, **MAX**, **MIN**, or **SUM**, an aggregate function that manipulates the retrieved data can also be used in the *expression*. As the aggregate function returns only one result, you cannot specify a general column which has not been grouped by an aggregate function in the **SELECT** column list.

- *table_name.* * : Specifying the table name and using * has the same effect as specifying all columns for the given table.
- *variable* : The data retrieved by the *select_expression* can be saved in more than one variables.
- [:]*identifier* : By using the :identifier after **TO** (or **INTO**), you can save the data to be retrieved in the ':identifier' variable.

Example 1

The following is an example of retrieving host countries of the Olympic Games without any duplicates. This example is performed on the olympic table of demodb.

The **DISTINCT** or **UNIQUE** keyword allows only unique values in the query result set. For example, when there are multiple **olympic** records whose **host_nation** values are 'Greece', you can use such keywords to display only one value in the query result.

```
SELECT DISTINCT host_nation FROM olympic;
  host_nation
======================
  'Australia'
  'Belgium'
  'Canada'
  'Finland'
  'France'
...
```

Example 2

The following is an example that defines an alias to a column to be queried, and sorts the result record by using the column alias in the **ORDER BY** clause. At this time, the number of the result records is limited to 5 by using the **LIMIT** clause and FOR **ORDERBY_NUM()**.

```
SELECT host_year as col1, host_nation as col2 FROM olympic ORDER BY col2 LIMIT 5;
         col1  col2
===================================
         2000  'Australia'
         1956  'Australia'
         1920  'Belgium'
         1976  'Canada'
         1948  'England'

SELECT CONCAT(host_nation, ', ', host_city) AS host_place FROM olympic
ORDER BY host_place FOR ORDERBY_NUM() BETWEEN 1 AND 5;

  host_place
======================
  'Australia,  Melbourne'
  'Australia,  Sydney'
  'Belgium,  Antwerp'
  'Canada,  Montreal'
  'England,  London'
```

FROM Clause

General

Description

The **FROM** clause specifies the table in which data is to be retrieved in the query. If no table is referenced, the **FROM** clause can be omitted. Retrieval paths are as follows:

- Single table
- Subquery

- Derived table

Syntax

```
SELECT [ <qualifier> ] <select_expressions>
                          [ FROM <table_specification> [ {, <table specification>
| <join table specification> }... ]]


<select_expressions> ::= * | <expression_comma_list> | *, <expression_comma_list>

<table_specification> ::=
 <single_table_spec> [ <correlation> ] [ WITH (lock_hint) ] |
 <metaclass_specification> [ <correlation> ] |
 <subquery> <correlation> |
 TABLE ( <expression> ) <correlation>

<correlation> ::= [ AS ] <identifier> [ ( <identifier_comma_list> ) ]

<single_table_spec> ::= [ ONLY ] <table_name> |
                      ALL <table_name> [ EXCEPT <table_name> ]

<metaclass_specification> ::= CLASS <class_name>

lock_hint ::=
READ UNCOMMITTED
```

- *select_expressions* : One or more columns or expressions to query is specified. Use * to query all columns in the table. You can also specify an alias for a column or an expression to be queried by using the AS keyword. This keyword can be used in **GROUP BY**, **HAVING**, **ORDER BY** and **FOR** clauses. The position index of the column is given according to the order in which the column was specified. The starting value is 1.
- *table_specification* : At least one table name is specified after the **FROM** clause. Subqueries and derived tables can also be used in the **FROM** clause. For more information on subquery derived tables, see Subquery Derived Table.
- *lock_hint* : You can set **READ UNCOMMITTED** for the table isolation level. **READ UNCOMMITTED** is a level where dirty reads are allowed; see Transaction Isolation Level for more information on the CUBRID transaction isolation level.

Example

```
--FROM clause can be omitted in the statement
SELECT 1+1 AS sum_value;
    sum_value
=============
            2

--db_root can be used as a dummy table
SELECT 1+1 AS sum_value FROM db_root;
    sum_value
=============
            2

SELECT CONCAT('CUBRID', '2008' , 'R3.0') AS db_version;
  db_version
======================
  'CUBRID2008R3.0'
```

Derived Table

In the query statement, subqueries can be used in the table specification of the **FROM** clause. Such subqueries create derived tables where subquery results are treated as tables. A correlation specification must be used when a subquery that creates a derived table is used.

Derived tables are also used to access the individual element of an attribute that has a set value. In this case, an element of the set value is created as an instance in the derived table.

Subquery Derived Table

Description

Each instance in the derived table is created from the result of the subquery in the **FROM** clause. A derived table created form a subquery can have any number of columns and records.

Syntax

```
FROM (subquery) [ AS ] derived_table_name [( column_name [ {, column_name }_ ] )]
```

- The number of *column_name* and the number of columns created by the *subquery* must be identical.

Example 1

The following is an example of retrieving the sum of the number of gold medals won by Korea and that of silver medals won by Japan. This example shows a way of getting an intermediate result of the subquery and processing it as a single result, by using a derived table. The query returns the sum of the **gold** values whose **nation_code** is 'KOR' and the **silver** values whose **nation_code** column is 'JPN'.

```
SELECT SUM(n) FROM (SELECT gold FROM participant WHERE nation_code='KOR'
UNION ALL SELECT silver FROM participant WHERE nation_code='JPN') AS t(n);
  sum(n)
========
      82
```

Example 2

Subquery derived tables can be useful when combined with outer queries. For example, a derived table can be used in the **FROM** clause of the subquery used in the **WHERE** clause.

The following is a query example that shows **nation_code**, **host_year** and **gold** fields of the instances whose number of gold medals is greater than average sum of the number of silver and bronze medals when one or more sliver or bronze medals were won. In this example, the query (the outer **SELECT** clause) and the subquery (the inner **SELECT** clause) share the **nation_code** attribute.

```
SELECT nation_code, host_year, gold
FROM participant p
WHERE gold > ( SELECT AVG(s)
            FROM ( SELECT silver + bronze
            FROM participant
            WHERE nation_code = p.nation_code
            AND silver > 0
            AND bronze > 0
          ) AS t(s));
  nation_code          host_year          gold
=========================================
  'JPN'                    2004               16
  'CHN'                    2004               32
  'DEN'                    1996                4
  'ESP'                    1992               13
```

WHERE Clause

Description

In a query, a column can be processed based on conditions. The **WHERE** clause specifies a search condition for data.

Syntax

```
WHERE search_condition

search_condition :
• comparison_predicate
• between_predicate
• exists_predicate
• in_predicate
```

```
• null_predicate
• like_predicate
• quantified predicate
• set_predicate
```

The **WHERE** clause specifies a condition that determines the data to be retrieved by *search_condition* or a query. Only data for which the condition is true is retrieved for the query results. (**NULL** value is not retrieved for the query results because it is evaluated as unknown value.)

- *search_condition* : It is described in detail in the following sections.
- Basic Conditional Expression
- BETWEEN Conditional Expression
- EXISTS Conditional Expression
- IN Conditional Expression
- IS NULL Conditional Expression
- LIKE Conditional Expression
- ANY/SOME/aLL Conditional Expressions

The logical operator **AND** or **OR** can be used for multiple conditions. If **AND** is specified, all conditions must be true. If **OR** is specified, only one needs to be true. If the keyword **NOT** is preceded by a condition, the meaning of the condition is reserved. The following table shows the order in which logical operators are evaluated.

Priority	Operator	Function
1	**()**	Logical expressions in parentheses are evaluated first.
2	NOT	Negates the result of the logical expression.
3	AND	All conditions in the logical expression must be true.
4	OR	One of the conditions in the logical expression must be true.

GROUP BY ... HAVING Clause

Description

The **GROUP BY** clause is used to group the result retrieved by the **SELECT** statement based on a specific column. This clause is used to sort by group or to get the aggregation by group using the aggregation function. Herein, a group consists of records that have the same value for the column specified in the **GROUP BY** clause.

You can also set a condition for group selection by including the **HAVING** clause after the **GROUP BY** clause. That is, only groups satisfying the condition specified by the **HAVING** clause are queried out of all groups that are grouped by the **GROUP BY** clause.

By SQL standard, you cannot specify a column (hidden column) not defined in the **GROUP BY** clause to the SELECT column list. However, by using extended CUBRID grammars, you can specify the hidden column to the SELECT column list. If you do not use the extended CUBRID grammars, the **only_full_group_by** parameter should be set to **yes**. For more information, see Statement/Type-Related Parameters.

Syntax

```
SELECT ...
GROUP BY { col_name | expr | position } [ ASC | DESC ],... [ WITH ROLLUP ][ ORDER BY NULL ]
                        [ HAVING < search_condition> ]
```

- *col_name* | *expr* | *position* : Specify one or more column names, expressions, aliases or column location. Items are separated by commas. Columns are sorted on this basis.
- [**ASC**| **DESC**] : Specify the **ASC** or **DESC** sorting option after the columns specified in the **GROUP BY** clause. If the sorting option is not specified, the default value is **ASC**.
- *search_condition* : Specify the search condition in the **HAVING** clause. In the **HAVING** clause you can refer to the hidden columns not specified in the **GROUP BY** clause as well as to columns and aliases specified in the **GROUP BY** clause and columns used in aggregate functions.

- **WITH ROLLUP** : If you specify the **WITH ROLLUP** modifier in the **GROUP BY** clause, the aggregate information of the result value of each GROUPed BY column is displayed for each group, and the total of all result rows is displayed at the last row.
- **ORDER BY NULL** : You can avoid the sorting overhead caused by **GROUP BY** by specifying the **ORDER BY NULL** modifier in the GROUP BY clause.

Example

```
--creating a new table
CREATE TABLE sales_tbl
(dept_no int, name VARCHAR(20) PRIMARY KEY, sales_month int, sales_amount int DEFAULT 100);
INSERT INTO sales_tbl VALUES
(201, 'George' , 1, 450),
(201, 'Laura'  , 2, 500),
(301, 'Max'    , 4, 300),
(501, 'Stephan', 4, DEFAULT),
(501, 'Chang'  , 5, 150),
(501, 'Sue'    , 6, 150),
(NULL, 'Yoka'   ,4, NULL);

--selecting rows grouped by dept_no with ORDER BY NULL modifier
SELECT dept_no, avg(sales_amount) FROM sales_tbl
GROUP BY dept_no ORDER BY NULL;
      dept_no  avg(sales_amount)
================================
         NULL               NULL
          201                475
          301                300
          501                133

--conditions in WHERE clause operate first before GROUP BY
SELECT dept_no, avg(sales_amount) FROM sales_tbl
WHERE sales_amount > 100 GROUP BY dept_no;
      dept_no  avg(sales_amount)
================================
          201                475
          301                300
          501                150

--conditions in HAVING clause operate last after GROUP BY
SELECT dept_no, avg(sales_amount) FROM sales_tbl
WHERE sales_amount > 100 GROUP BY dept_no HAVING avg(sales_amount) > 200;
      dept_no  avg(sales_amount)
================================
          201                475
          301                300

--selecting and sorting rows with using column alias
SELECT dept_no AS a1, avg(sales_amount) AS a2 FROM sales_tbl
WHERE sales_amount > 200 GROUP BY a1 HAVING a2 > 200 ORDER BY a2;
           a1            a2
==========================
          301          300
          201          475

--selecting rows grouped by dept_no with WITH ROLLUP modifier
SELECT dept_no AS a1, name AS a2, avg(sales_amount) AS a3 FROM sales_tbl
WHERE sales_amount > 100 GROUP BY a1,a2 WITH ROLLUP;
           a1  a2                             a3
================================================
          201  'George'                      450
          201  'Laura'                       500
          201  NULL                          475
          301  'Max'                         300
          301  NULL                          300
          501  'Chang'                       150
          501  'Sue'                         150
          501  NULL                          150
         NULL  NULL                          310
```

ORDER BY Clause

Description

The **ORDER BY** clause sorts the query result set in ascending or descending order. If you do not specify a sorting option such as **ASC** or **DESC**, the result set in ascending order by default. If you do not specify the **ORDER BY** clause, the order of records to be queried may vary depending on query.

Syntax

```
SELECT ...
ORDER BY {col_name | expr | position} [ASC | DESC],...]
    [ FOR <orderby_for_condition> ] ]

<orderby_for_condition> ::=
ORDERBY_NUM() { BETWEEN int AND int } |
    { { = | =< | < | > | >= } int } |
    IN ( int, ...)
```

- *col_name* | *expr* | *position* : Specify an column name, expression, alias, or column location. One or more column names, expressions or aliases can be specified. Items are separated by commas. A column that is not specified in the list of **SELECT** columns can be specified.
- [**ASC**| **DESC**] : **ASC** means sorting in ascending order, and **DESC** is sorting in descending order. If the sorting option is not specified, the default value is **ASC**.

Example

```
--selecting rows sorted by ORDER BY clause
SELECT * FROM sales_tbl ORDER BY dept_no DESC, name ASC;
      dept_no  name                  sales_month  sales_amount
==============================================================
          501  'Chang'                         5           150
          501  'Stephan'                       4           100
          501  'Sue'                           6           150
          301  'Max'                           4           300
          201  'George'                        1           450
          201  'Laura'                         2           500
         NULL  'Yoka'                          4          NULL

--sorting reversely and limiting result rows by LIMIT clause
SELECT dept_no AS a1, avg(sales_amount) AS a2 FROM sales_tbl
GROUP BY a1 ORDER BY a2 DESC LIMIT 0,3;
           a1           a2
==========================
          201          475
          301          300
          501          133

--sorting reversely and limiting result rows by FOR clause
SELECT dept_no AS a1, avg(sales_amount) AS a2 FROM sales_tbl
GROUP BY a1 ORDER BY a2 DESC FOR ORDERBY_NUM() BETWEEN 1 AND 3;
           a1           a2
==========================
          201          475
          301          300
          501          133
```

LIMIT Clause

Description

The **LIMIT** clause can be used to limit the number of records displayed. It takes one or two arguments. You can specify a very big integer for *row_count* to output to the last row, starting from a specific row.

The **LIMIT** clause can be used as a prepared statement. In this case, the bind parameter (?) can be used instead of an argument.

INST_NUM() and **ROWNUM** cannot be included in the **WHERE** clause in a query that contains the **LIMIT** clause. Also, **LIMIT** cannot be used together with FOR **ORDERBY_NUM()** or **HAVING GROUPBY_NUM()**.

Syntax

```
LIMIT [offset,] row_count
```

- *offset* : Specify the offset value of the starting row to be output. The offset value of the starting row of the result set is 0; it can be omitted and the default value is 0.
- *row_count* : Specify the number of records to be output. You can specify an integer greater than 0.

Example

```
--LIMIT clause can be used in prepared statement
PREPARE STMT FROM 'SELECT * FROM sales tbl LIMIT ?, ?';
EXECUTE STMT USING 0, 10;

--selecting rows with LIMIT clause
SELECT * FROM sales_tbl WHERE sales_amount > 100 LIMIT 5;
      dept_no  name                  sales_month  sales_amount
==============================================================
          201  'George'                        1           450
          201  'Laura'                         2           500
          301  'Max'                           4           300
          501  'Chang'                         5           150
          501  'Sue'                           6           150

--LIMIT clause can be used in subquery
SELECT t1.* FROM
(SELECT * FROM sales_tbl AS t2 WHERE sales_amount > 100 LIMIT 5) AS t1 LIMIT 1,3;
      dept_no  name                  sales_month  sales_amount
==============================================================
          201  'Laura'                         2           500
          301  'Max'                           4           300
          501  'Chang'                         5           150
```

Outer Join

Description

A join is a query that combines the rows of two or more tables or virtual tables (views). In a join query, a condition that compares the columns that are common in two or more tables is called a join condition. Rows are retrieved from each joined table, and are combined only when they satisfy the specified join condition.

A join query using an equality operator (=) is called an equi-join, and one without any join condition is called a cartesian product. Meanwhile, joining a single table is called a self join. In a self join, table **ALIAS** is used to distinguish columns, because the same table is used twice in the **FROM** clause.

A join that outputs only rows that satisfy the join condition from a joined table is called an inner or a simple join, whereas a join that outputs both rows that satisfy and do not satisfy the join condition from a joined table is called an outer join. An outer join is divided into a left outer join which outputs all rowss of the left table as the result, a right outer join which outputs all rowss of the right table as the result and a full outer join which outputs all rows of both tables. If there is no column value that corresponds to a table on one side in the result of an outer join query, all rowss are returned as **NULL**.

Syntax

```
FROM table_specification [{, table_specification | join_table_specification}...]

table_specification :
table_specification [ correlation ]
CLASS table_name [ correlation ]
subquery correlation
TABLE (expression) correlation

join_table_specification :
```

```
[ INNER | {LEFT | RIGHT} [ OUTER ] ] JOIN table_specification
join_condition

join_condition :
ON search_condition
```

- *oin_table_specification*
- { **LEFT** | **RIGHT** } [**OUTER**] **JOIN** : **LEFT** is used for a left outer join query, and **RIGHT** is for a right outer join query.

CUBRID does not support full outer joins. Path expressions that include subqueries and sub-columns cannot be used in the join conditions of an outer join.

Join conditions of an outer join are specified in a different way from those of an inner join. In an inner join, join conditions are expressed in the **WHERE** clause; in an outer join, they appear after the **ON** keyword in the **FROM** clause. Other retrieval conditions can be used in the **WHERE** or **ON** clause, but the retrieval result can differ depending on whether the condition is used in the **WHERE** or **ON** clause.

The table execution order is fixed according to the order specified in the **FROM** clause. Therefore, when using an outer join, you should create a query statement in consideration of the table order. It is recommended to use standard statements using { **LEFT** | **RIGHT** } [**OUTER**] **JOIN**, because using an Oracle-style join query statements by specifying an outer join operator (**+**) in the **WHERE** clause, even if possible, might lead the execution result or plan in an unwanted direction.

Example 1

The following is an example of retrieving the years and host countries of the Olympic Games since 1950 where a world record has been set. The following query retrieves instances whose values of the **host_year** column in the **history** table are greater than 1950.

```
SELECT DISTINCT h.host_year, o.host_nation FROM history h, olympic o
WHERE h.host_year=o.host_year AND o.host_year>1950;
    host_year  host_nation
===================================
         1968  'Mexico'
         1980  'U.S.S.R.'
         1984  'United States of America'
         1988  'Korea'
         1992  'Spain'
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
```

Example 2

The following is an example of retrieving the years and host countries of the Olympic Games since 1950 where a world record has been set, but including the Olympic Games where any world records haven't been set in the result. This example can be expressed in the following right outer join query. In this example, all instances whose values of the **host_year** column in the **history** table are not greater than 1950 are also retrieved. All instances of **host_nation** are included because this is a right outer join. **host_year** that does not have a value is represented as **NULL**.

```
SELECT DISTINCT h.host_year, o.host_nation
FROM history h RIGHT OUTER JOIN olympic o ON h.host_year=o.host_year WHERE
o.host_year>1950;
    host_year  host_nation
===================================
         NULL  'Australia'
         NULL  'Canada'
         NULL  'Finland'
         NULL  'Germany'
         NULL  'Italy'
         NULL  'Japan'
         1968  'Mexico'
         1980  'U.S.S.R.'
         1984  'United States of America'
         1988  'Korea'
         1992  'Spain'
```

```
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
```

Example 3

A right outer join query can be converted to a left outer join query by switching the position of two tables in the **FROM** clause. The right outer join query in the previous example can be expressed as a left outer join query as follows:

```
SELECT DISTINCT h.host_year, o.host_nation
FROM olympic o LEFT OUTER JOIN history h ON h.host_year=o.host_year WHERE o.host_year>1950;
    host_year  host_nation
===================================
         NULL  'Australia'
         NULL  'Canada'
         NULL  'Finland'
         NULL  'Germany'
         NULL  'Italy'
         NULL  'Japan'
         1968  'Mexico'
         1980  'U.S.S.R.'
         1984  'United States of America'
         1988  'Korea'
         1992  'Spain'
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
14 rows selected.
```

In this example, **h.host_year=o.host_year** is an outer join condition, and **o.host_year > 1950** is a search condition. If the search condition is used not in the **WHERE** clause but in the **ON** clause, the meaning and the result will be different. The following query also includes instances whose values of **o.host_year** are not greater than 1950.

```
SELECT DISTINCT h.host_year, o.host_nation
FROM olympic o LEFT OUTER JOIN history h ON h.host_year=o.host_year AND
o.host_year>1950;

=== <Result of SELECT Command in Line 3> ===
    host_year  host_nation
===================================
         NULL  'Australia'
         NULL  'Belgium'
         NULL  'Canada'
...
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
```

Example 4

Outer joins can also be represented by using **(+)** in the **WHERE** clause. The above example is a query that has the same meaning as the example using the **LEFT OUTER JOIN**. The **(+)** syntax is not ISO/aNSI standard, so it can lead to ambiguous situations. It is recommended to use the standard syntax **LEFT OUTER JOIN** (or **RIGHT OUTER JOIN**) if possible.

```
SELECT DISTINCT h.host_year, o.host_nation FROM history h, olympic o
WHERE o.host_year=h.host_year(+) AND o.host_year>1950;
    host_year  host_nation
===================================
         NULL  'Australia'
         NULL  'Canada'
         NULL  'Finland'
         NULL  'Germany'
         NULL  'Italy'
         NULL  'Japan'
         1968  'Mexico'
         1980  'U.S.S.R.'
         1984  'United States of America'
         1988  'Korea'
         1992  'Spain'
         1996  'United States of America'
```

```
         2000  'Australia'
         2004  'Greece'
```

Subquery

A subquery can be used wherever expressions such as **SELECT** or **WHERE** clause can be used. If the subquery is represented as an expression, it must return a single column; otherwise it can return multiple rows. Subqueries can be divided into single-row subquery and multiple-row subquery depending on how they are used.

Single-Row Subquery

Description

A single-row subquery outputs an row that has a single column. If no row is returned by the subquery, the subquery expression has a **NULL** value. If the subquery is supposed to return more than one rows, an error occurs.

Example

The following is an example of retrieving the history table as well as the host country where a new world record has been set. This example shows a single-row subquery used as an expression. In this example, the subquery returns **host_nation** values for the rows whose values of the **host_year** column in the **olympic** table are the same as those of the **host_year** column in the **history** table. If there are no values that meet the condition, the result of the subquery is **NULL**.

```
SELECT h.host_year, (SELECT host_nation FROM olympic o WHERE o.host_year=h.host_year),
h.event_code, h.score, h.unit from history h;
    host_year (SELECT host_nation FROM olympic o WHERE
o.host_year=h.host_year)   event_code  score  unit
==========================================================================================
====================
         2004  'Greece'                                                            20283
'07:53.0' 'time'
         2004  'Greece'                                                            20283
'07:53.0' 'time'
         2004  'Greece'                                                            20281
'03:57.0' 'time'
         2004  'Greece'                                                            20281
'03:57.0' 'time'
         2004  'Greece'                                                            20281
'03:57.0' 'time'
         2004  'Greece'                                                            20281
'03:57.0' 'time'
         2004  'Greece'                                                            20326
'210'     'kg'
         2000  'Australia'                                                         20328
'225'     'kg'
         2004  'Greece'                                                            20331
'237.5'   'kg'
...
```

Multiple-Row Subquery

Description

The multiple-row subquery returns one or more rows that contain the specified column. The result of the mutiple-row subquery can be used to create a set, a multiset or a list/sequence set using an appropriate keyword (**SET**, **MULTISET**, **LIST** or **SEQUENCE**).

Example

The following is an example of retrieving countries and their capital cities from the nation table, and returning lists of host countries and host cities of the Olympic Games. In this example, the subquery result is used to create a list from the values of the **host_city** column in the **olympic** table. This query returns **name** and **capital** value for **nation** table, as well as a set that contains **host_city** values of the **olympic** table with **host_nation** value. If the **name** value is an empty set in the query result, it is excluded. If there is no **olympic** table that has the same value as the **name**, an empty set is returned.

```
SELECT name, capital, list(SELECT host_city FROM olympic WHERE host_nation = name) FROM nation;
  name                   capital                sequence((SELECT host_city FROM olympic WHERE host_nation=name))
==================================================================
  'Somalia'              'Mogadishu'            {}
  'Sri Lanka'            'Sri Jayewardenepura Kotte'  {}
  'Sao Tome & Principe'  'Sao Tome'             {}
...
  'U.S.S.R.'             'Moscow'               {'Moscow'}
  'Uruguay'              'Montevideo'           {}
  'United States of America'  'Washington.D.C'      {'Atlanta ', 'St. Louis', 'Los Angeles', 'Los Angeles'}
  'Uzbekistan'           'Tashkent'             {}
  'Vanuatu'              'Port Vila'            {}
```

Such multiple-row subquery expressions can be used anywhere a set value expression is allowed. However, they cannot be used where a set constant value is required as in the **DEFAULT** specification in the class attribute definition.

If the **ORDER BY** clause is not used explicitly in the subquery, the order of the multiple-row query result is not set. Therefore, the order of the multiple-row subquery result that creates a sequence set must be specified by using the **ORDER BY** clause.

Hierarchical Query

START WITH ... CONNECT BY Clause

Description

This clause is used to obtain a set of data organized in a hierarchy. The **START WITH ... CONNECT BY** clause is used in combination with the **SELECT** clause in the following form.

Syntax

```
SELECT column_list
    FROM table_joins | tables
    [WHERE join_conditions and/or filtering_conditions]
    [START WITH condition]
    CONNECT BY [NOCYCLE] condition
```

START WITH Clause

The **START WITH** clause will filter the rows from which the hierarchy will start. The rows that satisfy the **START WITH** condition will be the root nodes of the hierarchy. If **START WITH** is omitted, then all the rows will be considered as root nodes.

Note If **START WITH** clause is omitted or the rows that satisfy the **START WITH** condition does not exist, all of rows in the table are considered as root nodes; which means that hierarchy relationship of sub rows which belong each root is searched. Therefore, some of results can be duplicate.

CONNECT BY [NOCYCLE] or PRIOR Operator

- **PRIOR** : The **CONNECT BY** condition is tested for a pair of rows. If it evaluates to true, the two rows satisfy the parent-child relationship of the hierarchy. We need to specify the columns that are used from the parent row and the columns that are used from the child row. We can use the **PRIOR** operator when applied to a column, which will refer to the value of the parent row for that column. If **PRIOR** is not used for a column, the value in the child row is used.
- **NOCYCLE** : In some cases, the resulting rows of the table joins may contain cycles, depending on the **CONNECT BY** condition. Because cycles cause an infinite loop in the result tree construction, CUBRID detects them and either returns an error doesn't expand the branches beyond the point where a cycle is found (if the

NOCYCLE keyword is specified).
This keyword may be specified after the **CONNECT BY** keywords. It makes CUBRID run a statement even if the processed data contains cycles.
If a **CONNECT BY** statement causes a cycle at runtime and the **NOCYCLE** keyword is not specified, CUBRID will return an error and the statement will be canceled. When specifying the **NOCYCLE** keyword, if CUBRID detects a cycle while processing a hierarchy node, it will set the **CONNECT_BY_ISCYCLE** attribute for that node to the value of 1 and it will stop further expansion of that branch.

Example

For the following samples, you will need the following structures:

Table tree

ID	MgrID	Name	BirthYear
1	NULL	KIM	1963
2	NULL	Moy	1958
3	1	Jonas	1976
4	1	Simth	1974
5	2	Verma	1973
6	2	Foster	1972
7	6	Brown	1981

Target tree_cycle

ID	MgrID	Name
1	NULL	Kim
2	11	Moy
3	1	Jonas
4	1	Smith
5	3	Verma
6	3	Foster
7	4	Brown
8	4	Lin
9	2	Edwin
10	9	Audrey
11	10	Stone

```
-- Creating tree table and then inserting data
CREATE TABLE tree(ID INT, MgrID INT, Name VARCHAR(32), BirthYear INT);

INSERT INTO tree VALUES (1,NULL,'Kim', 1963);
INSERT INTO tree VALUES (2,NULL,'Moy', 1958);
INSERT INTO tree VALUES (3,1,'Jonas', 1976);
INSERT INTO tree VALUES (4,1,'Smith', 1974);
INSERT INTO tree VALUES (5,2,'Verma', 1973);
INSERT INTO tree VALUES (6,2,'Foster', 1972);
INSERT INTO tree VALUES (7,6,'Brown', 1981);

-- Creating tree_cycle table and then inserting data
CREATE TABLE tree_cycle(ID INT, MgrID INT, Name VARCHAR(32));

INSERT INTO tree_cycle VALUES (1,NULL,'Kim');
INSERT INTO tree_cycle VALUES (2,11,'Moy');
INSERT INTO tree_cycle VALUES (3,1,'Jonas');
INSERT INTO tree_cycle VALUES (4,1,'Smith');
INSERT INTO tree_cycle VALUES (5,3,'Verma');
```

```
INSERT INTO tree_cycle VALUES (6,3,'Foster');
INSERT INTO tree_cycle VALUES (7,4,'Brown');
INSERT INTO tree_cycle VALUES (8,4,'Lin');
INSERT INTO tree_cycle VALUES (9,2,'Edwin');
INSERT INTO tree_cycle VALUES (10,9,'Audrey');
INSERT INTO tree_cycle VALUES (11,10,'Stone');

-- Executing a hierarchy query with CONNECT BY clause
SELECT id, mgrid, name
    FROM tree
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name
======================
1   null        Kim
2   null        Moy
3   1       Jonas
3   1       Jonas
4   1       Smith
4   1       Smith
5   2       Verma
5   2       Verma
6   2       Foster
6   2       Foster
7   6       Brown
7   6       Brown
7   6       Brown

-- Executing a hierarchy query with START WITH clause
SELECT id, mgrid, name
    FROM tree
    START WITH mgrid IS NULL
    CONNECT BY prior id=mgrid
    ORDER BY id;

id  mgrid       name
=============================
1   null        Kim
2   null        Moy
3   1       Jonas
4   1       Smith
5   2       Verma
6   2       Foster
7   6       Brown
```

Hierarchical Query for Table Join

Join Conditions

The table joins are evaluated first using the join conditions, if any. The conditions found in the **WHERE** clause are classified as join conditions or filtering conditions. All the conditions in the **FROM** clause are classified as join conditions. Only the join conditions are evaluated; the filtering conditions are kept for later evaluation. We recommended placing all join conditions in the **FROM** clause only so that conditions that are intended for joins are not mistakenly classified as filtering conditions.

Query Results

The resulting rows of the table joins are filtered according to the **START WITH** condition to obtain the root nodes for the hierarchy. If no **START WITH** condition is specified, then all the rows resulting from the table joins will be considered as root nodes.

After the root nodes are obtained, CUBRID will select the child rows for the root nodes. These are all nodes from the table joins that respect the **CONNECT BY** condition. This step will be repeated for the child nodes to determine their child nodes and so on until no more child nodes can be added.

In addition, CUBRID evaluates the **CONNECT BY** clause first and all the rows of the resulting hierarchy tress by using the filtering condition in the **WHERE** clause.

Example

The example illustrates how joins can be used in **CONNECT BY** queries. The joins are evaluated before the **CONNECT BY** condition and the join result will be the starting table on which the two clauses (**START WITH** clause and **CONNECT BY** clause).

```
-- Creating tree2 table and then inserting data
CREATE TABLE tree2(id int, treeid int, job varchar(32));

INSERT INTO tree2 VALUES(1,1,'Partner');
INSERT INTO tree2 VALUES(2,2,'Partner');
INSERT INTO tree2 VALUES(3,3,'Developer');
INSERT INTO tree2 VALUES(4,4,'Developer');
INSERT INTO tree2 VALUES(5,5,'Sales Exec.');
INSERT INTO tree2 VALUES(6,6,'Sales Exec.');
INSERT INTO tree2 VALUES(7,7,'Assistant');
INSERT INTO tree2 VALUES(8,null,'Secretary');

-- Executing a hierarchical query onto table joins
SELECT t.id,t.name,t2.job,level
    FROM tree t
        inner join tree2 t2 on t.id=t2.treeid
    START WITH t.mgrid is null
    CONNECT BY prior t.id=t.mgrid
    ORDER BY t.id;

id  name        job     level
================================================
1   Kim     Partner     1
2   Moy     Partner     1
3   Jonas       Developer   2
4   Smith       Developer   2
5   Verma       Sales Exec. 2
6   Foster      Sales Exec. 2
7   Brown       Assistant   3
```

Pseudo-Columns Available When Using the CONNECT BY Clause

LEVEL

This pseudo-column represents the level of the node in the hierarchy. Root nodes are considered to be at level 1, their children level 2 and so on.

The **LEVEL** pseudo-column may be used in the **SELECT** list, **WHERE** clause, **ORDER BY** clause, **GROUP BY ... HAVING** clauses and also in aggregate functions.

The following is an example of executing a hierarchical query with **LEVEL**.

```
-- Executing a hierarchical query with LEVEL
SELECT id, mgrid, name, LEVEL
    FROM tree
    WHERE LEVEL=2
    START WITH mgrid IS NULL
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;
id    mgrid        name        level
=========================================
3    1            Jonas        2
4    1            Smith        2
5    2            Verma        2
6    2            Foster       2
```

CONNECT_BY_ISLEAF

This pseudo-column indicates whether a hierarchical node is a leaf node or not. If the value for a row is 1, then the associated node is a leaf node.; otherwise, it will have the value 0 indicating that the node has children.

In this example, the **CONNECT_BY_ISLEAF** shows that the rows with the IDs 3, 4, 5 and 7 have no children.

```
-- Executing a hierarchical query with CONNECT_BY_ISLEAF
```

```
SELECT id, mgrid, name, CONNECT_BY_ISLEAF
      FROM tree
      START WITH mgrid IS NULL
      CONNECT BY PRIOR id=mgrid
      ORDER BY id;

id    mgrid        name        connect_by_isleaf
===========================================================
1    null          Kim         0
2    null          Moy         0
3    1             Jonas       1
4    1             Smith       1
5    2             Verma       1
6    2             Foster      0
7    6             Brown       1
```

CONNECT_BY_ISCYCLE

This pseudo-column indicates that a cycle was detected while processing the node, meaning that a child was also found to be an ancestor. A value of 1 for a row means a cycle was detected; the pseudo-column's value is 0, otherwise.

The **CONNECT_BY_ISCYCLE** pseudo-column may be used in the **SELECT** list, **WHERE** clause, **ORDER BY** clause, **GROUP BY** and **HAVING** clauses and also in aggregate functions (when the **GROUP BY** class exists in the statement).

Note This pseudo-column is available only when the **NOCYCLE** keyword is used in the statement.

The following is an example of executing a hierarchical query with **CONNECT_BY_ISCYCE** operator.

```
-- --Executing a hierarchical query with CONNECT_BY_ISCYCLE
SELECT id, mgrid, name, CONNECT_BY_ISCYCLE
    FROM tree_cycle
    START WITH name in ('Kim', 'Moy')
    CONNECT BY NOCYCLE PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name        connect_by_iscycle
==========================================================
1   null        Kim     0
2   11      Moy     0
3   1       Jonas       0
4   1       Smith       0
5   3       Verma       0
6   3       Foster      0
7   4       Brown       0
8   4       Lin     0
9   2       Edwin       0
10  9       Audrey      0
11  10      Stone       1
```

Operators Available When Using the CONNECT BY Clause

CONNECT_BY_ROOT Operator

This operator can be applied to columns and it returns the parent row or root row values for that column. This operator may be used in the **SELECT** list, **Where** clause and **ORDER BY** clause. When using the **CONNECT BY** clause some column operators become available.

The following is an example of executing a hierarchical query with **CONNECT_BY_ROOT** operator.

```
-- Executing a hierarchical query with CONNECT_BY_ROOT operator
SELECT id, mgrid, name, CONNECT_BY_ROOT id
    FROM tree
    START WITH mgrid IS NULL
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name        connect_by_root id
==========================================================
```

```
1   null        Kim     1
2   null        Moy     2
3   1       Jonas       1
4   1       Smith       1
5   2       Verma       2
6   2       Foster      2
7   6       Brown       2
```

PRIOR Operator

This operator may be applied to a column; it will return the parent node value for that column. For a root node, the operator will return the **NULL** value if it is applied to a column. This operator may be used in the **SELECT** list, **WHERE** clause, **ORDER BY** clause and also in the **CONNECT BY** clause.

The following is an example of executing a hierarchical query with **PRIOR** operator.

```
-- Executing a hierarchical query with PRIOR operator
SELECT id, mgrid, name, PRIOR id as "prior_id"
    FROM tree
    START WITH mgrid IS NULL
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name        prior_id
========================================
1   null        Kim     null
2   null        Moy     null
3   1       Jonas       1
4   1       Smith       1
5   2       Verma       2
6   2       Foster  2
7   6       Brown       6
```

Functions Available When Using the CONNECT BY Clause

Description

The **SYS_CONNECT_BY_PATH** function returns the branch of the node in the hierarchy. It returns a string that represents the concatenation of all the values obtained by evaluating the scalar expression for all the parents of a row, including that row, separated by the separator character, ordered ascending by level.

This function may be used in the **SELECT** list, **WHERE** clause and **ORDER BY** clause.

Syntax

```
SYS_CONNECT_BY_PATH (column_name, separator_char)
```

Example

The following is an example of executing a hierarchical query with **SYS_CONNECT_BY_PATH** function.

```
--Executing a hierarchical query with SYS_CONNECT_BY_PATH function
SELECT id, mgrid, name, SYS_CONNECT_BY_PATH(name,'/') as [hierarchy]
    FROM tree
    START WITH mgrid IS NULL
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name        hierarchy
=================================================
1   null        Kim     /Kim
2   null        Moy     /Moy
3   1       Jonas       /Kim/Jonas
4   1       Smith       /Kim/Smith
5   2       Verma       /Moy/Verma
6   2       Foster      /Moy/Foster
7   6       Brown       /Moy/Foster/Brown
```

Ordering Data with the Hierarchical Query

Description

The **ORDER SIBLINGS BY** clause will cause the ordering of the rows while preserving the hierarchy ordering so that the child nodes with the same parent will be stored according to the column list.

Syntax

```
ORDER SIBLINGS BY col_1 [ASC|DESC] [, col_2 [ASC|DESC] […[, col_n [ASC|DESC]]…]]
```

Example 1

The following example shows how to output information about seniors and subordinates in a company in the order of birth year.

The result with hierarchical query shows parent and child nodes in a row according to the column list specified in **ORDER SIBLINGS BY** statement by default. Sibling nodes that share the same parent node have outputted in a specified order.

```
-- Outputting a parent node and its child nodes, which sibling nodes that share the same
parent are sorted in the order of birth year.
SELECT id, mgrid, name, birthyear, level
FROM tree
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER SIBLINGS BY birthyear;

id        mgrid  name                    birthyear        level
==========================================================================
2         NULL  'Moy'                        1958            1
6            2  'Foster'                     1972            2
7            6  'Brown'                      1981            3
5            2  'Verma'                      1973            2
1         NULL  'Kim'                        1963            1
4            1  'Smith'                      1974            2
3            1  'Jonas'                      1976            2
```

Example 2

The following example shows how to output information about seniors and subordinates in a company in the order of joining. For the same level, the employee ID numbers are assigned in the order of joining. id indicates employee ID numbers (parent and child nodes) and mgrid indicates the employee ID numbers of their seniors.

```
-- Outputting siblings in a row
SELECT id, mgrid, name, LEVEL
FROM tree
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER SIBLINGS BY id;

id  mgrid       name        level
===============================================
1   null        Kim      1
3   1       Jonas       2
4   1       Smith       2
2   null        Moy      1
5   2       Verma       2
6   2       Foster      2
7   6       Brown       3
```

Scenario of Using Hierarchical Query

First of all let's start by giving a rough SQL translation of the **SELECT** statement with a **CONNECT BY** clause. For this we can consider that we have a table that contains a recurrent reference. We can consider that table to have two columns named ID and ParentID; ID is the primary key for the table and ParentID is a foreign-key to the same table. Naturally, the root nodes will have a ParentID value of **NULL**.

Now let us consider the fact that we want to get the full rows and a column with the level of the row in the hierarchy tree. For this we can write something similar to by querying with **UNION ALL**.

```
SELECT L1.ID, L1.ParentID, ..., 1 AS [Level]
    FROM tree_table AS L1
    WHERE L1.ParentID IS NULL
UNION ALL
SELECT L2.ID, L2.ParentID, ..., 2 AS [Level]
    FROM tree_table AS L1
        INNER JOIN tree_table AS L2 ON L1.ID=L2.ParentID
    WHERE L1.ParentID IS NULL
UNION ALL
SELECT L3.ID, L3.ParentID, ..., 3 AS [Level]
    FROM tree_table AS L1
        INNER JOIN tree_table AS L2 ON L1.ID=L2.ParentID
        INNER JOIN tree_table AS L3 ON L2.ID=L3.ParentID
    WHERE L1.ParentID IS NULL
UNION ALL ...
```

The problem with our approach is that we do not know how many levels we have. This could be rewritten in a stored procedure with a cycle until no new rows are retrieved, but we will have to check the tree for cycles at every step. Using a **SELECT** statement with a **CONNECT BY** clause we can rewrite this as follows.

This query will return the full hierarchy with the level of each row in the hierarchy.

```
SELECT ID, ParentID, ..., Level
    FROM tree_table
    START WITH ParentID IS NULL
    CONNECT BY ParentID=PRIOR ID
```

If we want to avoid the potential error caused by cycles we can write it as follows:

```
SELECT ID, ParentID, ..., Level
    FROM tree_table
    START WITH ParentID IS NULL
    CONNECT BY NOCYCLE ParentID=PRIOR ID
```

Performance of Hierarchical Query

Although this form is shorter and clearer, please keep in mind that it has its limitations regarding speed. If the result of the query contains all the rows of the table, the **CONNECT BY** form might be slower as it has to do additional processing (such as cycle detection, pseudo-column bookkeeping and others). However, if the result of the query only contains a part of the table rows, the **CONNECT BY** form might be faster.
For example, if we have a table with 20,000 records and we want to retrieve a sub-tree of roughly 1,000 records, a **SELECT** statement with a **START WITH ... CONNECT BY** clause will run up to 30% faster than an equivalent **UNION ALL** with **SELECT** statements.

INSERT

Overview

Description

You can insert a new record into a table in a database by using the **INSERT** statement. CUBRID supports **INSERT...VALUES**, **INSERT...SET** and **INSERT...SELECT** statements.

INSERT...VALUES and **INSERT...SET** statements are used to insert a new record based on the value that is explicitly specified while the **INSERT...SELECT** statement is used to insert query result records obtained from different tables. Use the **INSERT VALUES** or **INSERT...SELECT** statement to insert multiple rows by using the single **INSERT** statement.

Syntax

```
<INSERT … VALUES statement>
INSERT [INTO] table_name [(column_name, ...)]
    {VALUES | VALUE}({expr | DEFAULT}, ...)[,({expr | DEFAULT}, ...),...]
```

```
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
INSERT [INTO] table_name DEFAULT [ VALUES ]
INSERT [INTO] table_name VALUES()

<INSERT … SET statement>
INSERT [INTO] table_name
    SET column_name = {expr | DEFAULT}[, column_name = {expr | DEFAULT},...]
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]

<INSERT … SELECT statement>
INSERT [INTO] table_name [(column_name, ...)]
    SELECT...
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

- *table_name* : Specify the name of the target table into which you want to insert a new record.
- *column_name* : Specify the name of the column into which you want to insert the value. If you omit to specify the column name, it is considered that all columns defined in the table have been specified. Therefore, you must specify the values for all columns next to the **VALUES** keyword. If you do not specify all the columns defined in the table, a **DEFAULT** value is assigned to the non-specified columns; if the **DEFAULT** value is not defined, a **NULL** value is assigned.
- *expr* | **DEFAULT** : Specify values that correspond to the columns next to the **VALUES** keyword. Expressions or the **DEFAULT** keyword can be specified as a value. At this time, the order and number of the specified column list must correspond to the column value list. The column value list for a single record is described in parentheses.
- **DEFAULT** : You can use the **DEFAULT** keyword to specify a default value as the column value. If you specify **DEFAULT** in the column value list next to the **VALUES** keyword, a default value column is saved for the given column: if you specify **DEFAUL**T before the **VALUES** keyword, default values are saved for all columns in the table. **NULL** is saved for the column whose default value has not been defined.
- **ON DUPLICATE KEY UPDATE** : In case constraints are violated because a duplicated value for a column where **PRIMARY KEY** or **UNIQUE** attribute is defined is inserted, the value that makes constraints violated is changed into a specific value by performing the action specified in the **ON DUPLICATE KEY UPDATE** statement.

Example

```
CREATE TABLE a_tbl1(
id INT UNIQUE,
name VARCHAR,
phone VARCHAR DEFAULT '000-0000');

--insert default values with DEFAULT keyword before VALUES
INSERT INTO a_tbl1 DEFAULT VALUES;

--insert multiple rows
INSERT INTO a_tbl1 VALUES (1,'aaa', DEFAULT),(2,'bbb', DEFAULT);

--insert a single row specifying column values for all
INSERT INTO a_tbl1 VALUES (3,'ccc', '333-3333');

--insert two rows specifying column values for only
INSERT INTO a_tbl1(id) VALUES (4), (5);

--insert a single row with SET clauses
INSERT INTO a_tbl1 SET id=6, name='eee';
INSERT INTO a_tbl1 SET id=7, phone='777-7777';

SELECT * FROM a_tbl1;
           id  name                  phone
=========================================================
         NULL  NULL                  '000-0000'
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            4  NULL                  '000-0000'
            5  NULL                  '000-0000'
            6  'eee'                 '000-0000'
            7  NULL                  '777-7777'
```

INSERT ... SELECT Statement

Description

If you use the **SELECT** query in the **INSERT** statement, you can insert query results obtained from at least one table. The **SELECT** statement can be used in place of the **VALUES** keyword, or be included as a subquery in the column value list next to **VALUES**. If you specify the **SELECT** statement in place of the **VALUES** keyword, you can insert multiple query result records into the column of the table at once. However, there should be only one query result record if the **SELECT** statement is specified in the column value list.

In this way, you can extract data from another table that satisfies a certain retrieval condition, and insert it into the target table by combining the **SELECT** statement with the **INSERT** statement.

Syntax

```
INSERT [INTO] table_name [(column_name, ...)]
    SELECT...
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

Example

```
--creating an empty table which schema replicated from a_tbl1
CREATE TABLE a_tbl2 LIKE a_tbl1;

--inserting multiple rows from SELECT query results
INSERT INTO a_tbl2 SELECT * FROM a_tbl1 WHERE id IS NOT NULL;

--inserting column value with SELECT subquery specified in the value list
INSERT INTO a_tbl2 VALUES(8, SELECT name FROM a_tbl1 WHERE name <'bbb', DEFAULT);

SELECT * FROM a_tbl2;
           id  name                  phone
=========================================================
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            4  NULL                  '000-0000'
            5  NULL                  '000-0000'
            6  'eee'                 '000-0000'
            7  NULL                  '777-7777'
            8  'aaa'                 '000-0000'
```

ON DUPLICATE KEY UPDATE Statement

Description

In a situation in which a duplicate value is inserted into a column for which the **UNIQUE** index or the **PRIMARY KEY** constraint has been set, you can update to a new value without outputting the error by specifying the **ON DUPLICATE KEY UPDATE** clause in the **INSERT** statement.

However, the **ON DUPLICATE KEY UPDATE** clause cannot be used in a table in which a trigger for **INSERT** or **UPDATE** has been activated, or in a nested **INSERT** statement.

Syntax

```
<INSERT … VALUES statement>
<INSERT … SET statement>
<INSERT … SELECT statement>
    INSERT ...
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

- *column_name* = *expr* : Specifies the name of the column whose value you want to change next to **ON DUPLICATE KEY UPDATE** and a new column value by using the equal sign.

Example

```
--creating a new table having the same schema as a_tbl1
```

```
CREATE TABLE a_tbl3 LIKE a_tbl1;
INSERT INTO a_tbl3 SELECT * FROM a_tbl1 WHERE id IS NOT NULL and name IS NOT NULL;
SELECT * FROM a_tbl3;

           id  name                  phone
=========================================================
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            6  'eee'                 '000-0000'

--insert duplicated value violating UNIQUE constraint
INSERT INTO a_tbl3 VALUES(2, 'bbb', '222-2222');

ERROR: Operation would have caused one or more unique constraint violations.

--insert duplicated value with specifying ON DUPLICATED KEY UPDATE clause
INSERT INTO a_tbl3 VALUES(2, 'bbb', '222-2222')
ON DUPLICATE KEY UPDATE phone = '222-2222';

SELECT * FROM a_tbl3 WHERE id=2;
           id  name                  phone
=========================================================
            2  'bbb'                 '222-2222'
```

UPDATE

Description

You can update the column value of a record saved in the target table to a new one by using the **UPDATE** statement. Specify the name of the column to update and a new value in the **SET** clause, and specify the condition to be used to extract the record to be updated in the WHERE Clause. You can also specify the number of records to be updated in the **LIMIT** clause. You can use the update with the **ORDER BY** clause if you want to maintain the execution order or lock order of triggers.

Syntax

```
UPDATE table_name SET column_name = {expr | DEFAULT} [, column_name = {expr | DEFAULT}...]
    [WHERE search_condition]
    [ORDER BY {col_name | expr}]
    [LIMIT row_count]
```

- *table_name* : Specify the name of the table to be updated.
- *column_name* : Specify the columns to be updated.
- *expr* | **DEFAULT** : Specify a new value for the column, and specify an expression or the **DEFAULT** keyword as the value. You can also specify the **SELECT** query, which returns a single result record.
- *search_condition* : You can update the column value only for the record that satisfies the condition by specifying one in the WHERE Clause.
- *col_name* | *expr* : Specifies a column used as a basis for the update order.
- *row_count* : Specify the number of records to be updated after the LIMIT Clause. An integer greater than 0 can be specified.

Remark

One column can be updated only once in the same **UPDATE** statement.

Example

```
--creating a new table having all records copied from a_tbl1
CREATE TABLE a_tbl5 AS SELECT * FROM a_tbl1;
SELECT * FROM a_tbl5 WHERE name IS NULL;
           id  name                  phone
=========================================================
         NULL  NULL                  '000-0000'
            4  NULL                  '000-0000'
            5  NULL                  '000-0000'
```

```
            7  NULL                  '777-7777'

UPDATE a_tbl5 SET name='yyy', phone='999-9999' WHERE name IS NULL LIMIT 3;
SELECT * FROM a_tbl5;
           id  name                  phone
=========================================================
         NULL  'yyy'                 '999-9999'
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            4  'yyy'                 '999-9999'
            5  'yyy'                 '999-9999'
            6  'eee'                 '000-0000'
            7  NULL                  '777-7777'

-- using triggers, that the order in which the rows are updated is modified by the ORDER
BY clause.

CREATE TABLE t (i INT,d INT);
CREATE TRIGGER trigger1 BEFORE UPDATE ON t IF new.i < 10 EXECUTE PRINT 'trigger1 executed';
CREATE TRIGGER trigger2 BEFORE UPDATE ON t IF new.i > 10 EXECUTE PRINT 'trigger2 executed';
INSERT INTO t VALUES (15,1),(8,0),(11,2),(16,1), (6,0),(1311,3),(3,0);
UPDATE t  SET i = i + 1 WHERE 1 = 1;

trigger2 executed
trigger1 executed
trigger2 executed
trigger2 executed
trigger1 executed
trigger2 executed
trigger1 executed

TRUNCATE TABLE t;
INSERT INTO t VALUES (15,1),(8,0),(11,2),(16,1), (6,0),(1311,3),(3,0);
UPDATE t SET i = i + 1 WHERE 1 = 1  ORDER BY i;

trigger1 executed
trigger1 executed
trigger1 executed
trigger2 executed
trigger2 executed
trigger2 executed
trigger2 executed
```

REPLACE

Description

The **REPLACE** statement is working like **INSERT**, but the difference is that it inserts a new record after deleting the existing record without displaying the error when a duplicate value is inserted into a column for which **PRIMARY KEY** and **UNIQUE** constraints have defined. You must have both **INSERT** and **DELETE** privileges to use the **REPLACE** statement, because it performs insertion or insertion after deletion operations.

The **REPLACE** statement determines whether a new record causes the duplication of **PRIMARY KEY** or **UNIQUE** index column values. Therefore, for performance reasons, it is recommended to use the **INSERT** statement for a table for which a **PRIMARY KEY** or **UNIQUE** index has not been defined. The **REPLACE** statement is an extension of the SQL standard. See the following regarding the use of this statement.

- The **REPLACE** statement cannot contain subqueries.
- The **REPLACE** statement cannot be used for tables for which an **INSERT** or **DELETE** trigger has been set.
- An assignment statement such as **SET** *col_name* = *col_name* + 1 is not valid. Change such a statement to **SET** *col_name* = **DEFAULT**(*col_name*) + 1. Here, a non-NULL default value should be set for the *col_name* column.

Syntax

```
<REPLACE … VALUES statement>
REPLACE [INTO] table_name [(column_name, ...)]
    {VALUES | VALUE}({expr | DEFAULT}, ...)[,({expr | DEFAULT}, ...),...]
```

```
<REPLACE … SET statement>
REPLACE [INTO] table_name
    SET column_name = {expr | DEFAULT}[, column_name = {expr | DEFAULT},...]

<REPLACE … SELECT statement>
REPLACE [INTO] table_name [(column_name, ...)]
    SELECT...
```

- *table_name* : Specify the name of the target table into which you want to insert a new record.
- *column_name* : Specify the name of the column into which you want to insert the value. If you omit to specify the column name, it is considered that all columns defined in the table have been specified. Therefore, you must specify the value for the column next to **VALUES**. If you do not specify all the columns defined in the table, a **DEFAULT** value is assigned to the non-specified columns; if the **DEFAULT** value is not defined, a NULL value is assigned.
- *expr* | **DEFAULT** : Specify values that correspond to the columns after **VALUES**. Expressions or the **DEFAULT** keyword can be specified as a value. At this time, the order and number of the specified column list must correspond to the column value list. The column value list for a single record is described in parentheses.

Example

```
--creating a new table having the same schema as a_tbl1
CREATE TABLE a_tbl4 LIKE a_tbl1;
INSERT INTO a_tbl4 SELECT * FROM a_tbl1 WHERE id IS NOT NULL and name IS NOT NULL;
SELECT * FROM a_tbl4;
           id  name                  phone
=========================================================
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            6  'eee'                 '000-0000'

--insert duplicated value violating UNIQUE constraint
REPLACE INTO a_tbl4 VALUES(1, 'aaa', '111-1111'),(2, 'bbb', '222-2222');
REPLACE INTO a_tbl4 SET id=6, name='fff', phone=DEFAULT;

SELECT * FROM a_tbl4;
           id  name                  phone
=========================================================
            3  'ccc'                 '333-3333'
            1  'aaa'                 '111-1111'
            2  'bbb'                 '222-2222'
            6  'fff'                 '000-0000'
```

DELETE

Description

You can delete records in the table by using the **DELETE** statement. You can specify delete conditions by combining the statement with the WHERE Clause . If you want to limit the number of records to be deleted, you can do so by specifying the number of records to be deleted after the LIMIT Clause. In this case, only **row_count** records are deleted even when the number of records satisfying the WHERE Clause exceeds **row_count**.

Syntax

```
DELETE FROM <table_specification> [ WHERE <search_condition> ] [LIMIT row_count]

<table_specification> ::= <table_hierarchy> | ( <table_hierarchy_comma_list )

<table_hierarchy> ::= [ ONLY ] <table_name> |
                      ALL <table_name> [ EXCEPT <table_specification> ]
```

- *table_name* : Specifies the name of the table that contains the data to be deleted.
- *search_condition* : Delete only the data that meets the *search_condition* by using the WHERE Clause. If it is not specified, all the data in the table will be deleted.
- *row_count* : Specify the number of records to be deleted after the LIMIT Clause. An integer greater than 0 can be specified.

Example

```
CREATE TABLE a_tbl(
id INT NOT NULL,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333'), (4, NULL), (5,
NULL);

DELETE FROM a_tbl WHERE phone IS NULL LIMIT 1;

--delete one record only from a_tbl
SELECT * FROM a_tbl;

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            5  NULL

--delete all records from a_tbl
DELETE FROM a_tbl;
```

TRUNCATE

Description

You can delete all records in the specified table by using the **TRUNCATE** statement.

This statement internally delete first all indexes and constraints defined in a table and then deletes all records. Therefore, it performs the job faster than using the **DELETE FROM** *table_name* statement without a **WHERE** clause.

If the **PRIMARY KEY** constraint is defined in the table and this is referred by one or more **FOREIGN KEY**, it follows the **FOREIGN KEY ACTION**. If the **ON DELETE** action of **FOREIGN KEY** is **RESTRICT** or **NO_ACTION**, the **TRUNCATE** statement returns an error. If it is **CASCADE**, it deletes **FOREIGN KEY**. The **TRUNCATE** statement initializes the **AUTO INCREMENT** column of the table. Therefore, if data is inserted, the **AUTO INCREMENT** column value increases from the initial value.

Syntax

```
TRUNCATE [ TABLE ] <table_name>
```

- *table_name* : Specify the name of the table that contains the data to be deleted.

Example

```
CREATE TABLE a_tbl(A INT AUTO_INCREMENT(3,10) PRIMARY KEY);
INSERT INTO a_tbl VALUES (NULL),(NULL),(NULL);
SELECT * FROM a_tbl;
            a
=============
            3
            13
            23

--AUTO_INCREMENT column value increases from the initial value after truncating the table
TRUNCATE TABLE a_tbl;
INSERT INTO a_tbl VALUES (NULL);
SELECT * FROM a_tbl;
            a
=============
            3
```

DO

Description

The **DO** statement executes the specified expression, but does not return the result. Generally, the execution speed of the **DO** statement is higher than that of the **SELECT** expression statement, because the database server does not return the operation result or errors.

Syntax

```
DO expression
```

- *expression* : Specify an expression.

PREPARED STATEMENT

Overview

In general, the prepared statement is executed through the interface functions of JDBC, PHP, or ODBC; it can also be executed in the SQL level. The following SQL statements are provided for execution of prepared statement.

- Prepare the SQL statement to execute.

```
PREPARE stmt_name FROM preparable_stmt
```

- Execute the prepared statement.

```
EXECUTE stmt_name [USING value [, value] …]
```

- Drop the prepared statement.

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

PREPARE Statement

Description

The **PREPARE** statement prepares the query specified in *preparable_stmt* of the **FROM** clause, and assigns the name to be used later when the SQL statement is referenced to *stmt_name*. See EXECUTE Statement for example.

```
PREPARE stmt_name FROM preparable_stmt
```

- *stmt_name* : The prepared statement is specified. If an SQL statement with the same *stmt_name* exists in the given client session, clear the existing prepared statement and prepare a new SQL statement. If the **PREPARE** statement is not executed properly due to an error in the given SQL statement, it is processed as if the *stmt_name* assigned to the SQL statement does not exist.
- *preparable_stmt* : You must use only one SQL statement. Multiple SQL statements cannot be specified. You can use a question mark (?) as a bind parameter in the *preparable_stmt* statement and it should not be enclosed with quotes.

Caution

PREPARE statements are started by connecting an application to a server and will be maintained until the application terminates the connection. The connection maintained during this period is called a session. You can set the session time with the **session_state_timeout** parameter of **cubrid.conf**; the default value is **21600** seconds (=6 hours).

The data managed by the session includes **PREPARE** statements, custom variables, the last ID inserted(**LAST_INSERT_ID**) and the number of rows affected by the statement that you execute at the end(**ROW_COUNT**).

EXECUTE Statement

Description

The **EXECUTE** statement executes the prepared statement. You can bind the data value after the **USING** clause if a bind parameter (?) is included in the prepared statement. You cannot specify user-defined variables like an attribute in the **USING** clause. An value such as literal and an input parameter only can be specified.

Syntax

```
EXECUTE stmt_name [USING value [, value] …]
```

- *stmt_name* : The name given to the prepared statement to be executed is specified. An error message is displayed if the *stmt_name* is not valid, or if the prepared statement does not exist.
- *value* : The data to bind is specified if there is a bind parameter in the prepared statement. The number and the order of the data must correspond to that of the bind parameter. If it does not, an error message is displayed.

Example

```
PREPARE st FROM 'SELECT 1 + ?';
EXECUTE st USING 4;
   1+ ?:0
==========================
   5

SET @a=3;
EXECUTE st USING @a;
   1+ ?:0
==========================
   4

PREPARE st FROM 'SELECT ? + ?';
EXECUTE st USING 1,3;
   ?:0 + ?:1
==========================
   4

PREPARE st FROM 'SELECT ? + ?';
EXECUTE st USING 'a','b';
   ?:0 + ?:1
==========================
   'ab'

PREPARE st FROM 'SELECT FLOOR(?)';
EXECUTE st USING '3.2';
   floor( ?:0 )
==========================
   3.000000000000000e+000

PREPARE st FROM 'SELECT FLOOR(?)';
EXECUTE st USING 3.2;
   floor( ?:0 )
==========================
   3.0
```

DEALLOCATE PREPARE, DROP PREPARE Statement

Description

DEALLOCATE PREPARE and **DROP PREPARE** are used interchangeably and they clear the prepared statement. All prepared statements are cleared automatically by the server when the client session is terminated even if the **DEALLOCATE PREPARE** or **DROP PREPARE** statement is not executed.

Syntax

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

- *stmt_name* : The name given to the prepared statement to be cleared is specified. An error message is displayed if the *stmt_name* is not valid, or if the prepared statement does not exist.

Example

```
DEALLOCATE PREPARE stmt1;
```

SET

Description

The **SET** statement is the syntax that specifies custom variables and the method that you can use to save values.

You can create custom variables in two ways. One is to use the **SET** statement and the other is to use the assignment statement of custom variables within SQL statements. You can delete the custom variables that you defined with the **DEALLOCATE** or the **DROP** statements.

Custom variables are also called session variables as they are used for maintaining connections within one application. Custom variables are used within the part of a connection session, and the custom variables defined by an application can not be accessed by other applications. When an application terminates connections, all variables will be removed automatically. Custom variables are limited to twenty per connection session for an application. If you already have twenty custom variables and want to define a new custom variable, you must remove some variables with the **DROP VARIABLE** statement.

You can use custom variables in most SQL statements. If you define custom variables and refer to them in one statement, the sequence is not guaranteed. That is, if you refer to the variables specified in the **SELECT** list of the **HAVING**, **GROUP BY** or **ORDER BY** clause, you may not get the values in the sequence you expect. You can not also use custom variables as identifiers, such as column names or table names within SQL statements

Custom variables are not case-sensitive. The custom variable type can be one of the **SHORT**, **INTEGER**, **BIGINT**, **FLOAT**, **DOUBLE**, **NUMERIC**, **CHAR**, **VARCHAR**, **NCHAR**, **VARNCHAR**, **BIT** and **BIT VARYING**. Other types will be converted to the **VARCHAR** type.

```
SET @v1 = 1, @v2=CAST(1 AS BIGINT), @v3 = '123', @v4 = DATE'2010-01-01';

SELECT typeof(@v1), typeof(@v2), typeof(@v3), typeof(@v4);

   typeof(@v1)         typeof(@v2)         typeof(@v3)         typeof(@v4)
=================================================================================
  'integer'           'bigint'            'character var'     'character varying (10)'
```

Custom variables can be changed when you define values.

```
SELECT @v := 1, typeof(@v1), @v1:='1', typeof(@v1);

  @v := 1                typeof(@v1)          @v1 := '1'            typeof(@v1)
=================================================================================
  1                     'integer'            '1'                   'character (1)'
```

Syntax

```
<set_statement>
        : <set_statement>, <udf_assignment>
        | SET <udv_assignment>
        ;

<udv_assignment>
        : @<name> = <expression>
        | @<name> := <expression>
        ;

{DEALLOCATE|DROP} VARIABLE <variable_name_list>
<variable_name_list>
       : <variable_name_list> ',' @<name>
```

- You must define custom variable names with alphanumeric characters and underscores (_).
- When you define custom variables within SQL statements, you should use the ':=' operator.

Example

Define a custom variable, 'a' and assign a value, 1 to it.

```
SET @a = 1;

SELECT @a;

  @a
======================
  1
```

Use the custom variable to count the number of rows in the **SELECT** statement.

```
CREATE TABLE t (i INTEGER);
INSERT INTO t(i) VALUES(2),(4),(6),(8);

SET @a = 0;

SELECT @a := @a+1 AS row_no, i FROM t;

  row_no                          i
 ===================================
  1                               2
  2                               4
  3                               6
  4                               8
4 ROWS selected.
```

Use the custom variable as the input of the bind parameter defined in the prepared statement.

```
SET @a:=3;

PREPARE stmt FROM 'SELECT i FROM t WHERE i < ?';
EXECUTE stmt USING @a;

            i
=============
            2
```

Use the ':=' operator to define a custom variable.

```
SELECT @a := 1, @user_defined_variable := 'user defined variable';
UPDATE t SET i = (@var := 1);
```

Delete the custom variable a and user_defined_variable.

```
DEALLOCATE VARIABLE @a, @user_defined_variable;
DROP VARIABLE @a, @user_defined_variable;
```

Caution

The custom variables that are defined by the **SET** statement are started by connecting an application to a server and will be maintained until the application terminates the connection. The connection maintained during this period is called a session. When an application terminates the connection or when there are no requests for a certain period of time, the session will expire, and the custom variables will be deleted as a result. You can set the session time with the **session_state_timeout** parameter of **cubrid.conf**; the default value is **21600** seconds (=6 hours).

The data managed by the session includes **PREPARE** statements, custom variables, the last ID inserted(**LAST_INSERT_ID**) and the number of rows affected by the statement that you execute at the end(**ROW_COUNT**).

SHOW

SHOW TABLES Statement

Description

Displays the list of all the table names within a database. The name of the result column will be tables_in_<database name> and it will have one column. If you use the **LIKE** clause, you can search the table names matching this and if you use the **WHERE** clause, you can search table names with more general terms. **SHOW FULL TABLES** displays the second column, table_type together. The table must have the value, **BASE TABLE** and the view has the value, **VIEW**.

Syntax

```
SHOW [FULL] TABLES [LIKE 'pattern' | WHERE expr]
```

Example

The following is the result of executing the query with the demodb.

```
SHOW TABLES;
  Tables_in_demodb
======================
  'athlete'
  'code'
  'event'
  'game'
  'history'
  'nation'
  'olympic'
  'participant'
  'record'
  'stadium'

SHOW FULL TABLES;
  Tables_in_demodb     Table_type
============================================
  'athlete'             'BASE TABLE'
  'code'                'BASE TABLE'
  'event'               'BASE TABLE'
  'game'                'BASE TABLE'
  'history'             'BASE TABLE'
  'nation'              'BASE TABLE'
  'olympic'             'BASE TABLE'
  'participant'         'BASE TABLE'
  'record'              'BASE TABLE'
  'stadium'             'BASE TABLE'

SHOW FULL TABLES LIKE '%c%';
  Tables_in_demodb      Table_type
============================================
  'code'                'BASE TABLE'
  'olympic'             'BASE TABLE'
  'participant'         'BASE TABLE'
  'record'              'BASE TABLE'

SHOW FULL TABLES  WHERE table_type = 'BASE TABLE' and TABLES_IN_demodb LIKE
'%co%';   Tables_in_demodb      Table_type
============================================
  'code'                'BASE TABLE'
  'record'              'BASE TABLE'
```

SHOW COLUMN Statement

Description

Displays the column information of a table. You can use the **LIKE** clause to search the column names matching it. If you use the **WHERE** clause, you can search column names with more general terms like, "General Considerations for All **SHOW** Statements." If you use the **FULL** keyword, the additional information of a column will be displayed as follows:

- Field : Column name
- Type : Column data type
- Null : If you can store **NULL**, the value is YES and if not, it is NO
- Key : Whether a column has an index or not. If there is more than one key value in the given column of a table, this displays only the one that appears first in the order of PRI, UNI and MUL.

- If the key is a space, the column doesn't have an index, it is not the first column in the multiple column index or the index is non-unique.
- If the value is PRI, it is a primary key or the primary key of multiple columns.
- If the value is UNI, it is a unique index. (The unique index allows multiple NULL values but you can also set a NOT NULL constraint.)
- If the value is MUL, it is the first column of the non-unique index that allows the given value to be displayed in the column several times. If the column composes a composite unique index, the value will be MUL. The combination of column values can be unique but the value of each column can appear several times.
- Default : Default value defined in the column
- Extra : Additional information available about the given column. **AUTO_INCREMENT** The column attribute must have the auto_increment value.

SHOW FIELDS is the same command as **SHOW COLUMNS**.

The **DESCRIBE**(abbreviated **DESC**) statement and the **EXPLAIN** statement provide similar information to **SHOW COLUMNS**.

Syntax

```
SHOW COLUMNS {FROM | IN} tbl_name [LIKE 'pattern' | WHERE expr]
```

Example

The following is the result of a query for the demodb.

```
SHOW COLUMNS FROM athlete;
  Field                 Type                  Null                  Key
Default               Extra
==========================================================================================
==========================================
  'code'                'INTEGER'             'NO'                  'PRI'
NULL                  'auto_increment'
  'name'                'STRING(40)'          'NO'                  ''
NULL                  ''
  'gender'              'CHAR(1)'             'YES'                 ''
NULL                  ''
  'nation_code'         'CHAR(3)'             'YES'                 ''
NULL                  ''
  'event'               'STRING(30)'          'YES'                 ''
NULL                  ''

SHOW COLUMNS FROM athlete LIKE '%c%';
  Field                 Type                  Null                  Key
Default               Extra
==========================================================================================
==========================================
  'code'                'INTEGER'             'NO'                  'PRI'
NULL                  'auto_increment'
  'nation_code'         'CHAR(3)'             'YES'                 ''
NULL                  ''

SHOW COLUMNS FROM athlete  WHERE "type" = 'INTEGER' and "key"='PRI' AND
extra='auto_increment';
  Field                 Type                  Null                  Key
Default               Extra
==========================================================================================
==========================================
  'code'                'INTEGER'             'NO'                  'PRI'
NULL                  'auto_increment'
```

SHOW INDEX Statement

Description

The **SHOW INDEX** statement displays the index information. The query must have the following columns:

- Table : Table Name
- Non_unique

- 0 : Duplicate data are not allowed
- 1 : Duplicate data are allowed
- Key_name : Index name
- Seq_in_index : Serial number of the column in the index. Starts from 1.
- Column_name : Column name
- Collation :Method of sorting columns in the index. 'A' means ascending and **NULL** means not sorted.
- Cardinality : The numerical value of measuring the unique values in the index. Higher cardinality increases the opportunity of using an index. This value is updated every time **SHOW INDEX** is executed.
- Sub_part : The number of bytes of the indexed characters if the columns are indexed partially. **NULL** if all columns are indexed.
- Packed : Shows how keys are packed. If they are not packed, it will be **NULL**.
- Null : YES if a column can include **NULL**, NO if not.
- Index_type : Index to be used (currently, only the BTREE is supported.)

Syntax

```
SHOW {INDEX | INDEXES | KEYS } {FROM | IN} tbl_name
```

Example

The following is the result of a query for the demodb.

```
SHOW INDEX IN athlete;
   Table     Non_unique   Key_name       Seq_in_index  Column_name    Collation     Cardin
ality   Sub_part  Packed   Null   Index_type
==========================================================================================
================================================
 'athlete'     0      'pk_athlete_code'     1          'code'           'A'           6677
        NULL     NULL    'NO'      'BTREE'

CREATE TABLE t1( i1 INTEGER , i2 INTEGER NOT NULL, i3 INTEGER UNIQUE, s1 VARCHAR(10), s2
VARCHAR(10), s3 VARCHAR(10) UNIQUE);

CREATE INDEX i_t1_i1 ON t1(i1 desc);
CREATE INDEX i_t1_s1 ON t1(s1(7));
CREATE INDEX i_t1_i1_s1 ON t1(i1,s1);
CREATE UNIQUE INDEX i_t1_i2_s2 ON t1(i2,s2);

SHOW INDEXES FROM t1;
  Table  Non_unique  Key_name          Seq_in_index  Column_name   Collation   Cardinality
     Sub_part    Packed   Null    Index_type
==========================================================================================
================================================
  't1'          0  'i_t1_i2_s2'              1      'i2'          'A'            0
       NULL        NULL     'NO'    'BTREE'
  't1'          0  'i_t1_i2_s2'              2      's2'          'A'            0
       NULL        NULL     'YES'   'BTREE'
  't1'          0  'u_t1_i3'                 1      'i3'          'A'            0
       NULL        NULL     'YES'   'BTREE'
  't1'          0  'u_t1_s3'                 1      's3'          'A'            0
       NULL        NULL     'YES'   'BTREE'
  't1'          1  'i_t1_i1'                 1      'i1'          NULL           0
       NULL        NULL     'YES'   'BTREE'
  't1'          1  'i_t1_i1_s1'              1      'i1'          'A'            0
       NULL        NULL     'YES'   'BTREE'
  't1'          1  'i_t1_i1_s1'              2      's1'          'A'            0
       NULL        NULL     'YES'   'BTREE'
  't1'          1  'i_t1_s1'                 1      's1'          'A'            0
          7        NULL     'YES'   'BTREE'
```

SHOW GRANTS Statement

Description

The **SHOW GRANT** statement displays the permissions associated with the database user accounts.

Syntax

```
SHOW GRANTS FOR 'user'
```

Example

```
CREATE TABLE testgrant (id int);
CREATE USER user1;
GRANT INSERT,SELECT ON testgrant TO user1;

SHOW GRANTS FOR user1;
  Grants for USER1
======================
  'GRANT INSERT, SELECT ON testgrant TO USER1'
```

SHOW CREATE VIEW Statement

Description

If you specify a view name, the **SHOW CREATE VIEW** statement will display the corresponding **CREATE VIEW** statement.

Syntax

```
SHOW CREATE VIEW view_name
```

Example

The following is the result of a query for the demodb.

```
SHOW CREATE VIEW "db_class";
  View                Create View
========================================
  'db_class'         'SELECT c.class_name, CAST(c.owner.name AS VARCHAR(255)), CASE
c.class_type WHEN 0 THEN 'CLASS' WHEN 1 THEN 'VCLASS' ELSE
                   'UNKNOW' END, CASE WHEN MOD(c.is_system_class, 2) = 1 THEN 'YES' ELSE
'NO' END, CASE WHEN c.sub_classes IS NULL THEN 'NO'
                   ELSE NVL((SELECT 'YES' FROM _db_partition p WHERE p.class_of = c and
p.pname IS NULL), 'NO') END, CASE WHEN
                   MOD(c.is_system_class / 8, 2) = 1 THEN 'YES' ELSE 'NO' END FROM
_db_class c WHERE CURRENT_USER = 'DBA' OR {c.owner.name}
                   SUBSETEQ (  SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}),
SET{})  FROM db_user u, TABLE(groups) AS t(g)  WHERE
                   u.name = CURRENT_USER) OR {c} SUBSETEQ (  SELECT
SUM(SET{au.class_of})  FROM _db_auth au  WHERE {au.grantee.name} SUBSETEQ
                   (  SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})  FROM
db_user u, TABLE(groups) AS t(g)  WHERE u.name =
                   CURRENT_USER) AND  au.auth_type = 'SELECT')'
```

Transaction and Lock

Overview

This chapter covers issues relating to concurrency and restore, as well as how to commit or roll back transactions.

In multi-user environment, controlling access and update is essential to protect database integrity and ensure that a user's transaction will have accurate and consistent data. Without appropriate control, data could be updated incorrectly in the wrong order.

To control parallel operations on the same data, data must be locked during transaction, and unacceptable access to the data by another transaction must be blocked until the end of the transaction. In addition, any updates to a certain class must not be seen by other users before they are committed. If updates are not committed, all queries entered after the last commit or rollback of the update can be invalidated.

All examples introduced here were executed by csql. Outputs in the examples are displayed in *italics*.

Database Transaction

Overview

A database transaction groups CUBRID queries into a unit of consistency (for ensuring valid results in multi-user environment) and restore (for making the results of committed transactions permanent and ensuring that the aborted transactions are canceled in the database despite any failure, such as system failure). A transaction is a collection of one or more queries that access and update the database.

CUBRID allows multiple users to access the database simultaneously and manages accesses and updates to prevent inconsistency of the database. For example, if data is updated by one user, the changes made by this transaction are not seen to other users or the database until the updates are committed. This principle is important because the transaction can be rolled back without being committed.

You can delay permanent updates to the database until you are confident of the transaction result. Also, you can remove (**ROLLBACK**) all updates in the database if an unsatisfactory result or failure occurs in the application or computer system during the transaction. The end of the transaction is determined by the **COMMIT WORK** or **ROLLBACK WORK** statement. The **COMMIT WORK** statement makes all updates permanent while the **ROLLBACK WORK** statement cancels all updates entered in the transaction. For more information, see the Transaction Commit and Transaction Rollback sections.

Transaction Commit

Description

Updates that occurred in the database are not permanently stored until the **COMMIT WORK** statement is executed. "Permanently stored" means that storing the updates in the disk is completed; The **WORK** keyword can be omitted. In addition, other users of the database cannot see the updates until they are permanently applied. For example, when a new row is inserted into a class, only the user who inserted the row can access it until the database transaction is committed. (If the **UNCOMMITTED INSTANCES** isolation level is used, other users can see inconsistent uncommitted updates.)

All locks obtained by the transaction are released after the transaction is committed.

Syntax

```
COMMIT [ WORK ]
```

Example

The database transaction in the following example consists of three **UPDATE** statements and changes three column values of seats from the stadium. To compare the results, check the current values and names before the update is made. Since, by default, csql runs in an autocommit mode, the following example is executed after setting the autocommit mode to off.

```
;autocommit off
AUTOCOMMIT IS OFF
SELECT name, seats
FROM stadium WHERE code IN (30138, 30139, 30140);
   name                        seats
==================================
    'Athens Olympic Tennis Centre'         3200
    'Goudi Olympic Hall'         5000
    'Vouliagmeni Olympic Centre'         3400
```

Let each **UPDATE** statement have the current seats of each stadium. To verify whether the command is correctly executed, you can retrieve the columns related to the seats table.

```
UPDATE stadium
SET seats = seats + 1000
WHERE code IN (30138, 30139, 30140);

SELECT name, seats FROM stadium WHERE code in (30138, 30139, 30140);
    name                        seats
===================================
    'Athens Olympic Tennis Centre'         4200
    'Goudi Olympic Hall'         6000
    'Vouliagmeni Olympic Centre'         4400
```

If the update is properly done, the changes can be semi-permentanetly fixed. In this time, use the **COMMIT WORK** as below:

```
COMMIT WORK;
```

Note In CUBRID, an auto-commit mode is set by default for transaction management. An auto-commit mode is a mode that commits or rolls back all SQL statements. The transaction is committed automatically if the SQL is executed successfully, or is rolled back automatically if an error occurs.
Such auto commit modes are supported in CUBRID JDBC, ODBC, OLEDB and the CSQL Interpreter. In CUBRID CCI and CUBRID PHP, auto commit modes can be applied only for SELECT statements by setting broker parameters. For details, see Parameter by Broker. For a session command (;AUtocommit) that sets the auto-commit mode in the CSQL interpreter, see Session Commands.

Transaction Rollback

Description

The **ROLLBACK WORK** statement removes all updates to the database since the last transaction. The **WORK** keyword can be omitted. By using this statement, you can cancel incorrect or unnecessary updates before they are permanently applied to the database. All locks obtained during the transaction are released.

Syntax

```
ROLLBACK [ WORK ]
```

Example

The following example shows two commands that modify the definition and the row of the same table.

```
ALTER TABLE code DROP s_name;
INSERT INTO code (s_name, f_name) VALUES ('D','Diamond');

ERROR: s_name is not defined.
```

The **INSERT** statement fails because the s_name column has been dropped in the definition of code. The data intended to be entered to the code table is correct, but the s_name column is wrongly removed. At this point, you can use the **ROLLBACK WORK** statement to restore the original definition of the code table.

```
ROLLBACK WORK;
```

Later, remove the s_name column by entering the **ALTER TABLE** again and modify the **INSERT** statement. The **INSERT** command must be entered again because the transaction has been aborted. If the database update has been done as intended, commit the transaction to make the changes permanent.

```
ALTER TABLE code drop s_name;
INSERT INTO code (f_name) VALUES ('Diamond');

COMMIT WORK;
```

Savepoint and Partial Rollback

Description

A savepoint is established during the transaction so that database changes made by the transaction are rolled back to the specified savepoint. Such operation is called a partial rollback. In a partial rollback, database operations (insert, update, delete, etc.) after the savepoint are rolled back, and transaction operations before it are not rolled back. The transaction can proceed with other operations after the partial rollback is executed. Or the transaction can be terminated with the **COMMIT WORK** or **ROLLBACK WORK** statement. Note that the savepoint does not commit the changes made by the transaction.

A savepoint can be created at a certain point of the transaction, and multiple savepoints can be used for a certain point. If a partial rollback is executed to a savepoint before the specified savepoint or the transaction is terminated with the **COMMIT WORK** or **ROLLBACK WORK** statement, the specified savepoint is removed. The partial rollback after the specified savepoint can be performed multiple times.

Savepoints are useful because intermediate steps can be created and named to control long and complicated utilities. For example, if you use a savepoint during the update operation, you don't need to perform all statements again when you made a mistake.

Syntax 1

```
SAVEPOINT mark
mark:
_ a SQL identifier
_ a host variable (starting with :)
```

If you make *mark* all the same value when you specify multiple savepoints in a single transaction, only the latest savepoint appears in the partial rollback. The previous savepoints remain hidden until the rollback to the latest savepoint is performed and then appears when the latest savepoint disappears after being used.

Syntax 2

```
ROLLBACK [ WORK ] [ TO [ SAVEPOINT ] mark ] [ ]
mark:
_ a SQL identifier
_ a host variable (starting with :)
```

Previously, the **ROLLBACK WORK** statement canceled all database changes added since the latest transaction. The **ROLLBACK WORK** statement is also used for the partial rollback that rolls back the transaction changes after the specified savepoint.

If *mark* value is not given, the transaction terminates canceling all changes including all savepoints created in the transaction. If *mark* value is given, changes after the specified savepoint are canceled and the ones before it are remained.

Example

The following is an example of rolling back part of the transaction.

First, set savepoints SP1 and SP2.

```
CREATE TABLE athlete2 (name VARCHAR(40), gender CHAR(1), nation_code CHAR(3), event
VARCHAR(30));
```

```
INSERT INTO athlete2(name, gender, nation_code, event)
VALUES ('Lim Kye-Sook', 'W', 'KOR', 'Hockey');
SAVEPOINT SP1;

SELECT * from athlete2;
INSERT INTO athlete2(name, gender, nation_code, event)
VALUES ('Lim Jin-Suk', 'M', 'KOR', 'Handball');

SELECT * FROM athlete2;
SAVEPOINT SP2;

RENAME TABLE athlete2 AS sportsman;
SELECT * FROM sportsman;
ROLLBACK WORK TO SP2;
```

In the example above, the name change of the athlete2 table is rolled back by the partial rollback. The following is an example of executing the query with the original name and examining the result.

```
SELECT * FROM athlete2;
DELETE FROM athlete2 WHERE name = 'Lim Jin-Suk';
SELECT * FROM athlete2;
ROLLBACK WORK TO SP2;
```

The following is an example of rolling back to SP1.

```
SELECT * FROM athlete2;
ROLLBACK WORK TO SP1;
SELECT * FROM athlete2;
COMMIT WORK;
```

Database Concurrency

If there are multiple users with read and write privileges in a database, possibility exists that more than one user will access the database simultaneously. Controlling access and update in multi-user environment is essential to protect database integrity and ensure that users and transactions should have accurate and consistent data. Without appropriate control, data could be updated incorrectly in the wrong order.

Like most commercial database systems, CUBRID adopts serializability, an element that is essential to maintaining data concurrency within the database. Serializability ensures no interference between transactions when multiple transactions are executed at the same time. It is guaranteed more with the higher isolation level. This principle is based on the assumption that database consistency is guaranteed as long as transaction is executed automatically. This will be covered in the Lock Protocol section in detail.

The transaction must ensure database concurrency, and each transaction must guarantee appropriate results. When multiple transactions are being executed at the same time, an event in transaction T1 should not affect an event in transaction T2. This means isolation. Transaction isolation level is the degree to which a transaction is separated from all other concurrent transactions. The higher isolation level means the lower interference from other transactions. The lower isolation level means the higher the concurrency. A database determines whether which lock is applied to tables and records based on these isolation levels. Therefore, can control the level of consistency and concurrency specific to a service by setting appropriate isolation level.

You can set an isolation level by using the SET TRANSACTION ISOLATION LEVEL statement or system parameters provided by CUBRID. For more information, see Concurrency/Lock Parameters.

The read operations that allow interference between transactions with isolation levels are as follows:

- **Dirty read** : A transaction T2 can read D' before a transaction T1 updates data D to D' and commits it.
- **Non-repeatable read** : A transaction T1 can read other value, if a transaction T2 updates data while data is retrieved in the transaction T2 multiple times.
- **Phantom read** : A transaction T1 can read E, if a transaction T2 inserts new record E while data is retrieved in the transaction T1 multiple times.

The default value of CUBRID isolation level is REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES (3).

Isolation Levels Provided by CUBRID

CUBRID Isolation Level(isolation_level)	Other DBMS Isolation Level (isolation_level)	DIRTY READ	UNREPEATABLE READ	PHANTOM READ	Schema Changes o Table Being Retrie
SERIALIZABLE (6)	SERIALIZABLE (4)	N	N	N	N
REPEATABLE READ CLASS with REPEATABLE READ INSTANCES (5)	REPEATABLE READ (3)	N	N	Y	N
REPEATABLE READ CLASS with READ COMMITTED INSTANCES (4)	READ COMMITTED (2)	N	Y	Y	N
REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES (3)	READ UNCOMMITTED (1)	Y	Y	Y	N
READ COMMITTED CLASS with READ COMMITTED INSTANCES (2)		N	Y	Y	Y
READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES (1)		Y	Y	Y	Y

Lock Protocol

Overview

In the two-phase locking protocol used by CUBRID, a transaction obtains a shared lock before it reads an object, and an exclusive lock before it updates the object so that conflicting operations are not executed simultaneously.

If transaction T1 requires a lock, CUBRID checks if the requested lock conflicts with the existing one. If it does, transaction T1 enters a standby state and delays the lock. If another transaction T2 releases the lock, transaction T1 resumes and obtains it. Once the lock is released, the transaction do not require any more new locks.

Granularity Locking

CUBRID uses a granularity locking protocol to decrease the number of locks. In the granularity locking protocol, a database can be modeled as a hierarchy of lockable units: bigger locks have more granular locks.

For example, suppose that a database consists of multiple tables and each table consists of multiple instances. If the database is locked, all tables and instances are implicitly considered to be locked. A lock on a big unit results in less overhead, because only one lock needs to be managed. However, it leads to decreased concurrency because almost all concurrent transactions conflict with each other. The finer the granularity, the better the concurrency; it causes more overhead because more locks need to be managed. CUBRID selects a locking granularity level based on the operation being executed. For example, if a transaction retrieves all instances of a table, the entire tables will be locked, rather than each instance. If the transaction accesses a few instances of the table, the instances are locked individually.

If the locking granularities overlap, effects of a finer granularity are propagated in order to prevent conflicts. That is, if a shared lock is required on an instance of a table, an intention shared lock will be set on the table. If an exclusive lock is required on an instance of a table, an intention exclusive lock will be set on the table. An intention shared lock on a table means that a shared lock can be set on an instance of the table. An intention exclusive lock on a table means that a shared/exclusive lock can be set on an instance of the table. That is, if an intention shared lock on a table is allowed in one transaction, another transaction cannot obtain an exclusive lock on the table (for example, to add a new column). However, the second transaction may obtain a shared lock on the table. If an intention exclusive lock on the table is allowed in one transaction, another transaction cannot obtain a shared lock on the table (for example, a query on an instance of the tables cannot be executed because it is being changed).

A mechanism called lock escalation is used to limit the number of locks being managed. If a transaction has more than a certain number of locks (a number which can be changed by the **lock_escalation** system parameter), the system begins to require locks at the next higher level of granularity. This escalates the locks to a coarser level of granularity.

CUBRID performs lock escalation when no transactions have a higher level of granularity in order to avoid a deadlock caused by lock conversion.

Lock Mode Types And Compatibility

CUBRID determines the lock mode depending on the type of operation to be performed by the transaction, and determines whether or not to share the lock depending on the mode of the lock preoccupied by another transaction. Such decisions concerning the lock are made by the system automatically. Manual assignment by the user is not allowed. To check the lock information of CUBRID, use the **cubrid lockdb** *db_name* command. For details, see Checking Lock Status.

- **Shared lock (shared lock, S_LOCK)** : This lock is obtained before the read operation is executed on the object. It can be obtained by multiple transactions for the same object.
  Transaction T1 obtains the shared lock first before it performs the read operation on a certain object X, and releases it immediately after it completes the operation even before transaction T1 is committed. Here, transaction T2 and T3 can perform the read operation on X concurrently, but not the update operation.
- **Exclusive lock (exclusive lock, X_LOCK)** : This lock is obtained before the update operation is executed on the object. It can only be obtained by one transaction.
  Transaction T1 obtains the exclusive lock first before it performs the update operation on a certain object X, and does not release it until transaction T1 is committed even after the update operation is completed. Therefore, transaction T2 and T3 cannot perform the read operation as well on X before transaction T1 releases the exclusive lock.
- **Update lock (update lock, U_LOCK)** : This lock is obtained when the read operation is executed in the expression before the update operation is performed.
  For example, when an UPDATE statement combined with a **WHERE** clause is executed, execute the operation by obtaining the update lock for each tuple and the exclusive lock only for the result tuples that satisfy the condition when performing index search or full scan search in the **WHERE** clause. The update lock is converted to an exclusive lock when the actual update operation is performed. It can be called a quasi-exclusive lock because it does not allow the read lock on the same object for another transaction.
- **Intention lock (intention lock)** : A lock that is set inherently in a higher-level object than X to protect the lock on the object X of a certain level.
  For example, when a shared lock is requested for a certain tuple, prevent a situation from occurring in which the table is locked by another transaction by setting the intention shared lock as well on the table at the higher level in hierarchy. Therefore, the intention lock is not set on tuples at the lowest level, but is set on higher-level objects. The types of intention locks are as follows:
- **Intention shared lock (intention shared lock, IS_LOCK)** : If the intention shared lock is set on the table, which is the higher-level object, as the result of the shared lock set on a certain tuple, another transaction cannot perform operations such as changing the schema of the table (e.g. adding a column or changing the table name) or updating all tuples. However updating some tuples or viewing all tuples is allowed.
- **Intention exclusive lock (intention exclusive lock, IX_LOCK)** : If the intention exclusive lock is set on the table, which is the higher-level object, as the result of the exclusive lock set on a certain tuple, another transaction cannot perform operations such as changing the schema of the table, updating or viewing all tuples. However updating some tuples is allowed.
- **Shared with intent exclusive (shared with intent exclusive, SIX_LOCK)** : This lock is set on the higher-level object inherently to protect the shared lock set on all objects at the lower hierarchical level and the intention exclusive lock on some object at the lower hierarchical level.
  Once the shared intention exclusive lock is set on a table, another transaction cannot change the schema of the table, update all/some tuples or view all tuples. However, viewing some tuples is allowed.

The following table briefly shows the lock compatibility between the locks described below. Compatibility means that the lock requester can obtain a lock while the lock holder is keeping the lock obtained for the object X. N/a means 'not applicable'.

Lock Compatibility

		Lock Holder(lock holder)						
		NULL_LOCK	**IS_LOCK**	**S_LOCK**	**IX_LOCK**	**SIX_LOCK**	**U_LOCK**	**X_LOCK**
Lock Requester (lock requester)	**NULL_LOCK**	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
	IS_LOCK	TRUE	TRUE	TRUE	TRUE	TRUE	N/a	FALSE

	S_LOCK	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
	IX_LOCK	TRUE	TRUE	FALSE	TRUE	FALSE	N/a	FALSE
	SIX_LOCK	TRUE	TRUE	FALSE	FALSE	FALSE	N/a	FALSE
	U_LOCK	TRUE	N/a	TRUE	N/a	N/a	FALSE	FALSE
	X_LOCK	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

- **NULL_LOCK** : No lock

Example

session 1

```
;autocommit off
AUTOCOMMIT IS OFF
set transaction isolation level 4;
Isolation level set to:
REPEATABLE READ SCHEMA, READ COMMITTED
INSTANCES.
```

session 2

```
;autocommit off
AUTOCOMMIT IS OFF
set transaction
isolation level 4;
Isolation level
set to:
REPEATABLE READ
SCHEMA, READ
COMMITTED
INSTANCES.

/*
C:\CUBRID>cubrid
lockdb demodb

*** Lock Table
Dump ***
…

Object Lock Table:
        Current
number of objects
which are
locked    = 0
        Maximum
number of objects
which can be
locked = 10000
…
*/
```

session 1

```
SELECT nation_code, gold FROM participant
WHERE nation_code='USA';
 nation_code                  gold
======================================
'USA'                          36
'USA'                          37
'USA'                          44
'USA'                          37
'USA'                          36

/*
C:\CUBRID>cubrid lockdb demodb
*** Lock Table Dump ***
…

Object type: Root class.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
=  IS_LOCK, Count =   1, Nsubgranules =  1

Object type: Class = participant.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
=  IS_LOCK, Count =   2, Nsubgranules =  0
*/
```

<table>
<tr><td></td><td><pre>UPDATE participant
SET gold = 11
WHERE nation_code
= 'USA' ;</pre></td></tr>
<tr><td><pre>SELECT nation_code, gold FROM participant
WHERE nation_code='USA';

/* no results until transaction 2 releases a
lock

C:\CUBRID>cubrid lockdb demodb
*** Lock Table Dump ***
…

Object type: Instance of class
(0| 551| 7) = participant.
LOCK HOLDERS:
    Tran_index =   3, Granted_mode
=   X_LOCK, Count =   2

…

Object type: Root class.
LOCK HOLDERS:
    Tran_index =   3, Granted_mode
= IX_LOCK, Count =   1, Nsubgranules = 3
NON_2PL_RELEASED:
    Tran_index =   2, Non_2_phase_lock
= IS_LOCK

…

Object type: Class = participant.
LOCK HOLDERS:
    Tran_index =   3, Granted_mode
= IX_LOCK, Count =   3, Nsubgranules = 5
    Tran_index =   2, Granted_mode
= IS_LOCK, Count =   2, Nsubgranules = 0
*/</pre></td><td></td></tr>
<tr><td></td><td><pre>COMMIT;

Current
transaction has
been committed.</pre></td></tr>
<tr><td><pre>nation_code   gold
=================================
'USA'   11
'USA'   11
'USA'   11
'USA'   11
'USA'   11

/*
C:\CUBRID>cubrid lockdb demodb
…

Object type: Root class.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
= IS_LOCK, Count =   1, Nsubgranules = 1

Object type: Class = participant.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
= IS_LOCK, Count =   3, Nsubgranules = 0
…
*/</pre></td><td></td></tr>
</table>

```
COMMIT;

Current transaction has been committed.

/*
C:\CUBRID>cubrid lockdb demodb
…

Object Lock Table:
        Current number of objects which are
locked    = 0
        Maximum number of objects which can
be locked = 10000
*/
```

Transaction Deadlock

A deadlock is a state in which two or more transactions wait at the same time for another transaction's lock to be released. CUBRID resolves the problem by rolling back one of the transactions, because transactions in a deadlock state will hinder the work of another transaction. The transaction to be rolled back is usually the transaction which has made the least updates, i.e. the one that started more recently. As soon as a transaction is rolled back, the lock held by the transaction is released and other transactions in a deadlock are permitted to proceed.

It is impossible to predict such deadlocks, but it is recommended that you reduce the range to which lock is applied by setting the index, shortening the transaction, or setting the transaction isolation level as low in order to decrease such occurrences.

Example

session 1	session 2
;autocommit off AUTOCOMMIT IS OFF set transaction isolation level 6; Isolation level set to: SERIALIZABLE	;autocommit off AUTOCOMMIT IS OFF set transaction isolation level 6; Isolation level set to: SERIALIZABLE

CREATE TABLE lock_tbl(host_year
integer, nation_code char(3));
INSERT INTO lock_tbl VALUES (2004,
'KOR');
INSERT INTO lock_tbl VALUES (2004,
'USA');
INSERT INTO lock_tbl VALUES (2004,
'GER');
INSERT INTO lock_tbl VALUES (2008,
'GER');
COMMIT;
 	
SELECT * FROM lock_tbl;
    host_year nation_code
===================================
         2004 'KOR'
         2004 'USA'
         2004 'GER'
         2008 'GER'	

	SELECT * FROM lock_tbl;
	    host_year nation_code
	===================================
	         2004 'KOR'
	         2004 'USA'
	         2004 'GER'
	         2008 'GER'

DELETE FROM lock_tbl WHERE host_year=2008;	

```
/* no result until transaction 2
releases a lock

C:\CUBRID>cubrid lockdb demodb
*** Lock Table Dump ***
…

Object type: Class = lock_tbl.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
=   S_LOCK, Count =   2,
Nsubgranules =  0

BLOCKED LOCK HOLDERS:
    Tran_index =   1, Granted_mode
=   S_LOCK, Count =   3,
Nsubgranules =  0
    Blocked_mode = SIX_LOCK
    Start_waiting_at = Fri Feb 12
14:22:58 2010
    Wait_for_nsecs = -1

*/
```

```
INSERT INTO lock_tbl VALUES (2004,
'AUS');
```

```
ERROR: Your transaction (index 1,
dba@ 090205|4760) has been
unilaterally aborted by the system.

/*
System rolled back the transaction
1 to resolve a deadlock

C:\CUBRID>cubrid lockdb demodb
*** Lock Table Dump ***

Object type: Class = lock_tbl.

LOCK HOLDERS:
    Tran_index =   2, Granted_mode
= SIX_LOCK, Count =   3,
Nsubgranules =  0
*/
```

Transaction Timeout

CUBRID provides the lock timeout feature, which sets the waiting time for the lock until the transaction lock setting is allowed.

If the lock is allowed within the lock timeout, CUBRID rolls back the transaction and outputs an error message when the timeout has passed. If a transaction deadlock occurs within the lock timeout, CUBRID rolls back the transaction whose waiting time is closest to the timeout.

Setting the Lock Timeout

Description

The system parameter **lock_timeout_in_secs** in the **$CUBRID/conf/cubrid.conf** file or the **SET TRANSACTION** statement sets the timeout (in seconds) during which the application will wait for the lock and rolls back the transaction and outputs an error message when the specified time has passed. The default value of the **lock_timeout_in_secs** parameter is **-1**, which means the application will wait indefinitely until the transaction lock is allowed. Therefore, the user can change this value depending on the transaction pattern of the application. If the lock timeout value has been set to 0, an error message will be displayed as soon as a lock occurs.

Syntax

```
SET TRANSACTION LOCK TIMEOUT timeout_spec [ ; ]
timeout_spec:
- INFINITE
- OFF
- unsigned_integer
- variable
```

- **INFINITE** : Wait indefinitely until the transaction lock is allowed. Has the same effect as setting the system parameter **lock_timeout_in_secs**to -1.
- **OFF** : Do not wait for the lock, but roll back the transaction and display an error message. Has the same effect as setting the system parameter **lock_timeout_in_secs**to 0.
- *unsigned_integer* : Set in seconds. Wait for the transaction lock for the specified time period.
- *variable* : A variable can be specified. Wait for the transaction lock for the value saved by the variable.

Example 1

```
vi $CUBRID/conf/cubrid.conf
…
lock_timeout_in_secs = 10
…
```

Example 2

```
csql> SET TRANSACTION LOCK TIMEOUT 10;
```

Checking the Lock Timeout

Description

You can check the lock timeout set for the current application by using the **GET TRANSACTION** statement, or save this value in a variable.

Syntax

```
GET TRANSACTION LOCK TIMEOUT [ { INTO | TO } variable ] [ ; ]
```

Example

```
csql> GET TRANSACTION LOCK TIMEOUT;

         Result
===============
  1.000000e+001
```

Lock Timeout Error Message

Displays the following message if lock timeout occurs in a transaction that was waiting for another transaction's lock to be released. To increase the level of detail of the information displayed in the lock timeout error message, see the description of lock_timeout_message_type in Concurrency/Lock Parameters .

```
ERROR: Your transaction (index 3, cub_user@cdbs006.cub|15668) timed out waiting
on    X_LOCK lock on instance 0|636|34 of class participant. You are waiting for
user(s)   to finish.
```

- Your transaction(index 3 …) : This means that the index of the transaction that was rolled back due to timeout while waiting for the lock is 3. The transaction index is a number that is sequentially assigned when the client connects to the database server. You can also check this number by executing the **cubrid lockdb** utility.
- (…cub_user@cdbs006.cub|15668) : cub_user is the login ID of the client and the part after @ is the name of the host where the client was running. The part after| is the process ID (PID) of the client.
- X_LOCK : This means the exclusive lock set on the object to perform data update. For details, see Lock Mode Types And Compatibility.
- Instance 0|636|34 of class participant : This means that **X_LOCK** has been set on a certain tuple in the table named participant and the OID (unique ID assigned to the given object) of the tuple is 0|636|34.

That is, the above lock error message can be interpreted as meaning that "Because another client is holding **X_LOCK** on a certain tuple in the participant table, transaction 3 which running on the host cdbs006.cub waited for the lock and was rolled back as the timeout has passed."

If you want to check the lock information of the transaction specified in the error message, you can do so by using the **cubrid lockdb** utility to search for the OID value (ex: 0|636|34) of a specific tuple where the **X_LOCK** is set currently to find the transaction ID currently holding the lock, the client program name and the process ID (PID). For details, see Checking Lock Status. You can also check the transaction lock information in the CUBRID Manager.

You can organize the transactions by checking uncommitted queries through the SQL log after checking the transaction lock information in the manner described above. For information on checking the SQL log, see Broker Log.

Also, you can force problematic transactions to quit by using the **cubrid killtran** utility. For details, see Killing Transactions.

Transaction Isolation Level

Overview

The transaction isolation level is determined based on how much interference occurs. The more isolation means the less interference from other transactions and more serializable. The less isolation means the more interference from other transactions and higher level of concurrency. You can control the level of consistency and concurrency specific to a service by setting appropriate isolation level.

Note A transaction can be restored in all supported isolation levels because updates are not committed before the end of the transaction.

SET TRANSACTION ISOLATION LEVEL

Description

You can set the level of transaction isolation by using **isolation_level** and the **SET TRANSACTION** statement in the **$CUBRID/conf/cubrid.conf**. The level of **REPEATABLE READ CLASS** and **READ UNCOMMITTED INSTANCES** are set by default, which indicates the level 3 through level 1 to 6. For more information, see Database Concurrency.

Syntax

```
SET TRANSACTION ISOLATION LEVEL isolation_level_spec [ ; ]
isolation_level_spec:
_ SERIALIZABLE
_ CURSOR STABILITY
_ isolation_level [ { CLASS | SCHEMA } [ , isolation_level INSTANCES ] ]
_ isolation_level [ INSTANCES [ , isolation_level { CLASS | SCHEMA } ] ]
_ variable
isolation_level:
_ REPEATABLE READ
_ READ COMMITTED
_ READ UNCOMMITTED
```

Example 1

```
vi $CUBRID/conf/cubrid.conf
…
isolation_level = 1
…

or

isolation_level = "TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE"
```

Example 2

```
csql> SET TRANSACTION ISOLATION LEVEL 1;

or

csql> SET TRANSACTION ISOLATION LEVEL READ COMMITTED CLASS,READ UNCOMMITTED INSTANCES;
```

The following table shows the isolation levels from 1 to 6. It consists of table schema (row) and isolation level. For the unsupported isolation level, see Unsupported Combination of Isolation Level.

Levels of Isolation Supported by CUBRID

Name	Description
SERIALIZABLE (6)	In this isolation level, problems concerning concurrency (e.g. dirty read, non-repeatable read, phantom read, etc.) do not occur.
REPEATABLE READ CLASS with REPEATABLE READ INSTANCES (5)	Another transaction T2 cannot update the schema of table A while transaction T1 is viewing table A. Transaction T1 may experience phantom read for the record R that was inserted by another transaction T2 when it is repeatedly retrieving a specific record.
REPEATABLE READ CLASS with READ COMMITTED INSTANCES (or CURSOR STABILITY) (4)	Another transaction T2 cannot update the schema of table A while transaction T1 is viewing table A. Transaction T1 may experience R read (non-repeatable read) that was updated and committed by another transaction T2 when it is repeatedly retrieving the record R.
REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES (3)	Default isolation level. Another transaction T2 cannot update the schema of table A while transaction T1 is viewing table A. Transaction T1 may experience R' read (dirty read) for the record that was updated but not committed by another transaction T2.
READ COMMITTED CLASS with READ COMMITTED INSTANCES (2)	Transaction T1 may experience A' read (non-repeatable read) for the table that was updated and committed by another transaction T2 while it is viewing table A repeatedly. Transaction T1 may experience R' read (non-repeatable read) for the record that was updated and committed by another transaction T2 while it is retrieving the record R repeatedly.
READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES (1)	Transaction T1 may experience A' read (non-repeatable read) for the table that was updated and committed by another transaction T2 while it is repeatedly viewing table A. Transaction T1 may experience R' read (dirty read) for the record that was updated but not committed by another transaction T2.

If the transaction level is changed in an application while a transaction is executed, the new level is applied to the rest of the transaction being executed. Therefore, some object locks that have already been obtained may be released during the transaction while the new isolation level is applied. For this reason, it is recommended that the transaction isolation level be modified when the transaction starts (after commit, rollback or system restart) because an isolation level which has already been set does not apply to the entire transaction, but can be changed during the transaction.

GET TRANSACTION ISOLATION LEVEL

Description

You can assign the current isolation level to *variable* by using the **GET TRANSACTION** statement. The following is a statement that verifies the isolation level. *variable*.

Syntax

```
GET TRANSACTION ISOLATION LEVEL [ { INTO | TO } variable ] [ ; ]
```

Example

```
csql> GET TRANSACTION ISOLATION LEVEL;
       Result
=============
  READ COMMITTED SCHEMA, READ UNCOMMITTED INSTANCES
```

SERIALIZABLE

The highest isolation level (6). Problems concerning concurrency (e.g. dirty read, non-repeatable read, phantom read, etc.) do not occur.

The following are the rules of this isolation level:

- Transaction T1 cannot read or modify the record being updated by another transaction T2.
- Transaction T1 cannot read or modify the record being viewed by another transaction T2.
- Another transaction T2 cannot insert a new record into table A while transaction T1 is retrieving the records of table A.

This isolation level uses a two-phase locking protocol for shared and exclusive lock: the lock is maintained until the transaction ends even after the operation has been executed.

Example

The following is an example that shows that another transaction cannot access the table or record while one transaction is reading or updating the object when the transaction level of the concurrent transactions is **SERIALIZABLE**.

session 1	session 2
;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 6; Isolation level set to: SERIALIZABLE	;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 6; Isolation level set to: SERIALIZABLE
--creating a table CREATE TABLE isol6_tbl(host_year integer, nation_code char(3)); INSERT INTO isol6_tbl VALUES (2008, 'AUS'); COMMIT;	

	--selecting records from the table
	SELECT * FROM isol6_tbl WHERE
	nation_code = 'AUS';
	    host_year nation_code
	===================================
	         2008 'AUS'

INSERT INTO isol6_tbl VALUES (2004, 'AUS'); /* unable to insert a row until the tran 2 committed */	
	COMMIT;
	SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; /* unable to select rows until tran

	1 committed */
COMMIT;	host_year nation_code =================================== 2008 'AUS' 2004 'AUS'
DELETE FROM isol6_tbl WHERE nation_code = 'AUS' and host_year=2008; /* unable to delete rows until tran 2 committed */	
	COMMIT;
	SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; /* unable to select rows until tran 1 committed */
COMMIT;	host_year nation_code =================================== 2004 'AUS'
ALTER TABLE isol6_tbl ADD COLUMN gold INT; /* unable to alter the table schema until tran 2 committed */	/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; host_year nation_code =================================== 2004 'AUS'
	COMMIT;
	SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; /* unable to access the table until tran_1 committed */
COMMIT;	host_year nation_code gold =================================== 2004 'AUS' NULL

REPEATABLE READ CLASS with REPEATABLE READ INSTANCES

A relatively high isolation level (5). A dirty or non-repeatable read does not occur, but a phantom read may.

The following are the rules of this isolation level:

- Transaction T1 cannot read or modify the record being updated by another transaction T2.
- Transaction T1 cannot read or modify the record being viewed by another transaction T2.
- Another transaction T2 can insert a new record into table A while transaction T1 is retrieving records of table A. However, transaction T1 and T2 cannot set the lock on the same record.

This isolation level uses a two-phase locking protocol.

Example

The following is an example that shows that phantom read may occur because another transaction can add a new record while one transaction is performing the object read when the transaction level of the concurrent transactions is **REPEATABLE READ CLASS** with **REPEATABLE READ INSTANCES**.

session 1	session 2
;autocommit off AUTOCOMMIT IS OFF	;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 5

	SET TRANSACTION ISOLATION LEVEL 5 ;xr Isolation level set to: REPEATABLE READ SCHEMA, REPEATABLE READ INSTANCES.	;xr Isolation level set to: REPEATABLE READ SCHEMA, REPEATABLE READ INSTANCES.
	--creating a table CREATE TABLE isol5_tbl(host_year integer, nation_code char(3)); CREATE UNIQUE INDEX on isol5_tbl(nation_code, host_year); INSERT INTO isol5_tbl VALUES (2008, 'AUS'); INSERT INTO isol5_tbl VALUES (2004, 'AUS'); COMMIT; ;xr	

		--selecting records from the table
		SELECT * FROM isol5_tbl WHERE nation_code='AUS';
		;xr
		
		=== <Result of SELECT Command> ===
		    host_year nation_code
		===================================
		         2004 'AUS'
		         2008 'AUS'
		
		2 rows selected.

	INSERT INTO isol5_tbl VALUES (2004, 'KOR'); INSERT INTO isol5_tbl VALUES (2000, 'AUS'); ;xr 2 rows affected. /* able to insert new rows only when locks are not conflicted */	
		SELECT * FROM isol5_tbl WHERE nation_code='AUS'; ;xr /* phantom read may occur when tran 1 committed */

	COMMIT;
	;xr
		=== <Result of SELECT Command> ===
		    host_year nation_code
		===================================
		         2000 'AUS'
		         2004 'AUS'
		         2008 'AUS'
		
		3 rows selected.

	DELETE FROM isol5_tbl WHERE nation_code = 'AUS' and host_year=2008; ;xrun /* unable to delete rows	

until tran 2 committed */	
1 rows affected. 1 command(s) successfully processed.	COMMIT; ;xr
	SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr /* unable to select rows until tran 1 committed */
COMMIT; ;xr	=== <Result of SELECT Command> === host_year nation_code =================================== 2000 'AUS' 2004 'AUS' 2 rows selected.
ALTER TABLE isol5_tbl ADD COLUMN gold INT; ;xr /* unable to alter the table schema until tran 2 committed */	
	/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2000 'AUS' 2004 'AUS' 2 rows selected.
1 command(s) successfully processed.	COMMIT; ;xr
	SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr /* unable to access the table until tran_1 committed */
COMMIT; ;xr	=== <Result of SELECT Command > === host_year nation_code gold =================================== 2000 'AUS' NULL 2004 'AUS' NULL 2 rows selected.

REPEATABLE READ CLASS with READ COMMITTED INSTANCES

A relatively low isolation level (4). A dirty read does not occur, but non-repeatable or phantom read may. That is, transaction T1 can read another value because insert or update by transaction T2 is allowed while transaction T1 is repeatedly retrieving one object.

The following are the rules of this isolation level:

- Transaction T1 cannot read the record being updated by another transaction T2.

- Transaction T1 can update/insert record to the table being viewed by another transaction T2.
- Transaction T1 cannot change the schema of the table being viewed by another transaction T2.

This isolation level uses a two-phase locking protocol for an exclusive lock. A shared lock on a row is released immediately after it is read; however, an intention lock on a table is released when a transaction terminates to ensure repeatable read on the schema.

Example

The following is an example that shows that a phantom or non-repeatable read may occur because another transaction can add or update a record while one transaction is performing the object read but repeatable read for the table schema update is ensured when the transaction level of the concurrent transactions is **REPEATABLE READ CLASS** with **READ COMMITTED INSTANCES**.

session 1	session 2

`;autocommit off`
`AUTOCOMMIT IS OFF`
 	
`SET TRANSACTION ISOLATION LEVEL 4;`
 	
`Isolation level set to:`
`REPEATABLE READ SCHEMA,`
`READ COMMITTED INSTANCES.`
	`;autocommit off`
	`AUTOCOMMIT IS OFF`
	
	`SET TRANSACTION ISOLATION LEVEL 4;`
	
	`Isolation level set to:`
	`REPEATABLE READ SCHEMA, READ`
	`COMMITTED INSTANCES.`

`--creating a table`
 	
`CREATE TABLE isol4_tbl(host_year integer, nation_code char(3));`
`INSERT INTO isol4_tbl VALUES (2008, 'AUS');`
 	
`COMMIT;`	

	`--selecting records from the table`
	`SELECT * FROM isol4_tbl;`
	`    host_year  nation_code`
	`===================================`
	`         2008  'AUS'`

`INSERT INTO isol4_tbl VALUES (2004, 'AUS');`
 	
`INSERT INTO isol4_tbl VALUES (2000, 'NED');`
 	
`/* able to insert new rows even if tran 2 uncommitted */`	

	`SELECT * FROM isol4_tbl;`
	
	`/* phantom read may occur when tran 1 committed */`

`COMMIT;`	`    host_year  nation_code`
	`===================================`
	`         2008  'AUS'`
	`         2004  'AUS'`
	`         2000  'NED'`

`INSERT INTO isol4_tbl VALUES (1994, 'FRA');`	

	`SELECT * FROM isol4_tbl;`
	
	`/* unrepeatable read may occur when`

	tran 1 committed */
DELETE FROM isol4_tbl WHERE nation_code = 'AUS' and host_year=2008; /* able to delete rows while tran 2 is selecting rows*/	
COMMIT;	host_year nation_code =================================== 2004 'AUS' 2000 'NED' 1994 'FRA'
ALTER TABLE isol4_tbl ADD COLUMN gold INT; /* unable to alter the table schema until tran 2 committed */	
	/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol4_tbl; host_year nation_code =================================== 2004 'AUS' 2000 'NED' 1994 'FRA'
	COMMIT;
	SELECT * FROM isol4_tbl; /* unable to access the table until tran_1 committed */
COMMIT;	host_year nation_code gold =================================== 2004 'AUS' NULL 2000 'NED' NULL 1994 'FRA' NULL

REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES

The default isolation of CUBRID (3). The concurrency level is high. A dirty, non-repeatable or phantom read may occur for the tuple, but repeatable read is ensured for the table. That is, transaction T2 can read an object while transaction T1 is updating one.

The following are the rules of this isolation level:

- Transaction T1 can read the record being updated by another transaction T2.
- Transaction T1 can update/insert record to the table being viewed by another transaction T2.
- Transaction T1 cannot change the schema of the table being viewed by another transaction T2.

This isolation level uses a two-phase locking protocol for an exclusive and update lock. However, the shared lock on the tuple is released immediately after it is retrieved. The intention lock on the table is released when the transaction ends to ensure repeatable reads.

Example

The following is an example that shows that another transaction can read dirty data uncommitted by one transaction but repeatable reads are ensured for table schema update when the transaction level of the concurrent transactions is **REPEATABLE READ CLASS** with **READ UNCOMMITTED INSTANCES**.

<table>
<tr><th>session 1</th><th>session 2</th></tr>
<tr><td><pre>;autocommit off
AUTOCOMMIT IS OFF

SET TRANSACTION ISOLATION
LEVEL 3;

Isolation level set to:
REPEATABLE READ SCHEMA,
READ UNCOMMITTED INSTANCES.</pre></td><td><pre>;autocommit off
AUTOCOMMIT IS OFF

SET TRANSACTION ISOLATION LEVEL 3;

Isolation level set to:
REPEATABLE READ SCHEMA, READ
UNCOMMITTED INSTANCES.</pre></td></tr>
<tr><td><pre>--creating a table

CREATE TABLE
isol3_tbl(host_year
integer, nation_code
char(3));
CREATE UNIQUE INDEX on
isol3_tbl(nation_code,
host_year);
INSERT INTO isol3_tbl
VALUES (2008, 'AUS');

COMMIT;</pre></td><td></td></tr>
<tr><td></td><td><pre>--selecting records from the table
SELECT * FROM isol3_tbl;
    host_year nation_code
===================================
         2008 'AUS'</pre></td></tr>
<tr><td><pre>INSERT INTO isol3_tbl
VALUES (2004, 'AUS');

INSERT INTO isol3_tbl
VALUES (2000, 'NED');

/* able to insert new rows
even if tran 2 uncommitted
*/</pre></td><td></td></tr>
<tr><td></td><td><pre>SELECT * FROM isol3_tbl;
    host_year nation_code
===================================
         2008 'AUS'
         2004 'AUS'
         2000 'NED'

/* dirty read may occur so that
tran_2 can select new rows
uncommitted by tran_1 */</pre></td></tr>
<tr><td><pre>ROLLBACK;</pre></td><td></td></tr>
<tr><td></td><td><pre>SELECT * FROM isol3_tbl;
    host_year nation_code
===================================
         2008 'AUS'

/* unrepeatable read may occur so
that selected results are different
*/</pre></td></tr>
<tr><td><pre>INSERT INTO isol3_tbl
VALUES (1994, 'FRA');

DELETE FROM isol3_tbl
WHERE nation_code = 'AUS'
and
host_year=2008;</pre></td><td></td></tr>
</table>

/* able to delete rows even if tran 2 uncommitted */	

	SELECT * FROM isol3_tbl;
	    host_year nation_code
	===================================
	         1994 'FRA'

ALTER TABLE isol3_tbl ADD COLUMN gold INT; /* unable to alter the table schema until tran 2 committed */	

	/* repeatable read is ensured while
	tran_1 is altering table schema */
	
	SELECT * FROM isol3_tbl;
	    host_year nation_code
	===================================
	         1994 'FRA'

	COMMIT;
	SELECT * FROM isol3_tbl;
COMMIT;	host_year nation_code gold =================================== 1994 'FRA' NULL

Note CUBRID flushes dirty data (or dirty instances) in the client buffers to the database (server) such as the following situations. For more information, see How to Handle Dirty Instances.

READ COMMITTED CLASS with READ COMMITTED INSTANCES

A relatively low isolation level (2). A dirty read does not occur, but non-repeatable or phantom read may occur. That is, this level is similar to **REPEATABLE READ CLASS** with **READ COMMITTED INSTANCES**(level 4) described above, but works differently for table schema. Non-repeatable read due to a table schema update may occur because another transaction T2 can change the schema of the table being viewed by the transaction T1.

The following are the rules of this isolation level:

- Transaction T1 cannot read the record being updated by another transaction T2.
- Transaction T1 can update/insert a record to the table being viewed by another transaction T2.
- Transaction T1 can change the schema of the table being viewed by another transaction T2.

This isolation level uses a two-phase locking protocol for an exclusive lock. However, non-repeatable read may occur because the shared lock on the tuple is released immediately after it is retrieved and the intention lock on the table is released immediately as well.

Example

The following is an example that shows that phantom or non-repeatable read for the record as well as for the table schema may occur because another transaction can add or update a new record while one transaction is performing the object read when the transaction level of the concurrent transactions is **READ COMMITTED CLASS** with **READ COMMITTED INSTANCES**.

session 1	session 2
;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 2 ;xr	;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 2 ;xr Isolation level set to:

	Isolation level set to: READ COMMITTED SCHEMA, READ COMMITTED INSTANCES.	READ COMMITTED SCHEMA, READ COMMITTED INSTANCES.
	--creating a table CREATE TABLE isol2_tbl(host_year integer, nation_code char(3)); CREATE UNIQUE INDEX on isol2_tbl(nation_code, host_year); INSERT INTO isol2_tbl VALUES (2008, 'AUS'); COMMIT; ;xr	
		--selecting records from the table SELECT * FROM isol2_tbl; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2008 'AUS' 1 rows selected.
	INSERT INTO isol2_tbl VALUES (2004, 'AUS'); INSERT INTO isol2_tbl VALUES (2000, 'NED'); ;xr 2 rows affected. /* able to insert new rows even if tran 2 uncommitted */	
		SELECT * FROM isol2_tbl; ;xr /* phantom read may occur when tran 1 committed */
	COMMIT; ;xr	=== <Result of SELECT Command> === host_year nation_code =================================== 2008 'AUS' 2004 'AUS' 2000 'NED' 3 rows selected.
	INSERT INTO isol2_tbl VALUES (1994, 'FRA'); ;xr 1 rows affected.	
		SELECT * FROM isol2_tbl; ;xr /* unrepeatable read may occur when tran 1 committed */
	DELETE FROM isol2_tbl	

<table>
<tr><td><pre>WHERE nation_code = 'AUS'
and
host_year=2008;
;xr

1 rows affected.

/* able to delete rows even
if tran 2 uncommitted */</pre></td><td></td></tr>
<tr><td><pre>COMMIT;
;xr</pre></td><td><pre>=== <Result of SELECT Command> ===
    host_year nation_code
===================================
         2004 'AUS'
         2000 'NED'
         1994 'FRA'

3 rows selected.</pre></td></tr>
<tr><td><pre>ALTER TABLE isol2_tbl
ADD COLUMN gold INT;
;xr

1 command(s) successfully
processed.

/* able to alter the table
schema even if tran 2 is
uncommitted yet*/</pre></td><td></td></tr>
<tr><td></td><td><pre>/* unrepeatable read may occur so
that result shows different schema
*/

SELECT * FROM isol2_tbl;
;xr</pre></td></tr>
<tr><td><pre>COMMIT;
;xr</pre></td><td><pre>=== <Result of SELECT Command > ===
host_year nation_code gold
===================================
  2004 'AUS'   NULL
  2000 'NED'   NULL
  1994 'FRA'   NULL

3 rows selected.</pre></td></tr>
</table>

READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES

The lowest isolation level (1). The concurrency level is the highest. A dirty, non-repeatable or phantom read may occur for the tuple and a non-repeatable read may occur for the table as well. Similar to **REPEATABLE READ CLASS** with **READ UNCOMMITTED INSTANCES**(level 3) described above, but works differently for the table schema. That is, non-repeatable read due to table schema update may occur because another transaction T2 can change the schema of the table being viewed by the transaction T1.

The following are the rules of this isolation level:

- Transaction T1 can read the record being updated by another transaction T2.
- Transaction T1 can update/insert record to the table being viewed by another transaction T2.
- Transaction T1 can change the schema of the table being viewed by another transaction T2.

This isolation level uses a two-phase locking protocol for an exclusive and update lock. However, the shared lock on the tuple is released immediately after it is retrieved. The intention lock on the table is released immediately after the retrieval as well.

Example

<table>
<tr><th>session 1</th><th>session 2</th></tr>
<tr><td><pre>;autocommit off</pre></td><td><pre>;autocommit off</pre></td></tr>
</table>

	AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 1; Isolation level set to: READ COMMITED SCHEMA, READ UNCOMMITTED INSTANCES.	AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 1; Isolation level set to: READ COMMITED SCHEMA, READ UNCOMMITTED INSTANCES.
	--creating a table CREATE TABLE isol1_tbl(host_year integer, nation_code char(3)); CREATE UNIQUE INDEX on isol1_tbl(nation_code, host_year); INSERT INTO isol1_tbl VALUES (2008, 'AUS'); COMMIT;	
		--selecting records from the table SELECT * FROM isol1_tbl; host_year nation_code =================================== 2008 'AUS'
	INSERT INTO isol1_tbl VALUES (2004, 'AUS'); INSERT INTO isol1_tbl VALUES (2000, 'NED'); /* able to insert new rows even if tran 2 uncommitted */	
		SELECT * FROM isol1_tbl; host_year nation_code =================================== 2008 'AUS' 2004 'AUS' 2000 'NED' /* dirty read may occur so that tran_2 can select new rows uncommitted by tran_1 */
	ROLLBACK;	
		SELECT * FROM isol1_tbl; host_year nation_code =================================== 2008 'AUS' /* unrepeatable read may occur so that selected results are different */
	INSERT INTO isol1_tbl VALUES (1994, 'FRA'); DELETE FROM isol1_tbl WHERE nation_code = 'AUS' and host_year=2008; /* able to delete rows while tran 2 is selecting	

`rows*/`	

	`SELECT * FROM isol1_tbl;`
	`    host_year  nation_code`
	`===================================`
	`         1994  'FRA'`

`ALTER TABLE isol1_tbl`
`ADD COLUMN gold INT;`
 	
`/* able to alter the table schema even if tran 2 is uncommitted yet*/`	

	`/* unrepeatable read may occur so that result shows different schema */`
	
	`SELECT * FROM isol1_tbl;`

`COMMIT;`	`host_year  nation_code  gold`
	`====================================`
	`  1994  'FRA'           NULL`

UPDATE INCONSISTENCY

In this isolation level, uncommitted updates may be lost, which makes a transaction unrestorable (cannot be rolled back) because the data are committed before the end of the transaction. CUBRID does not support this isolation level because this can cause the updates made by the user to be lost. However, if this isolation level is specified, CUBRID provides an appropriate level to the user application.

The following are the rules of this isolation level:

- A transaction does not overwrite an object being modified by another transaction.

Note A transaction can be restored in all supported isolation levels because updates are not committed before the end of the transaction.

Combination of Unsupported Isolation Level

You can set customized isolation levels by using the **SET TRANSACTION ISOLATION LEVE** statement. However, combinations of isolation levels below are not supported. If they are used, a system error message is shown up and an isolation level closest to the one specified is chosen.

The following are unsupported isolation levels. If table schema is changed while data is selected, unrepeatable read occurs; therefore, the combinations below are not supported.

- **READ COMMITTED CLASS** with **REPEATABLE READ INSTANCES**
- **READ UNCOMMITTED CLASS** with **REPEATABLE READ INSTANCES**

Neither are isolation levels below supported because updating a row by a transaction is not allowed while table schema is changed by other transaction.

- **READ UNCOMMITTED CLASS** with **READ COMMITTED INSTANCES**
- **READ UNCOMMITTED CLASS** with **READ UNCOMMITTED INSTANCES**

How to Handle Dirty Instance

CUBRID flushes dirty data (or dirty instances) in the client buffers to the database (server) such as the following situations. In additions to those, there can be more situations where flushes can be performed.

- Dirty data can be flushed to server when a transaction is committed.
- Some of dirty data can be flushed to server when a lot of data is loaded into the client buffers.
- Dirty data of table A can be flushed to server when the schema of table A is updated.
- Dirty data of table A can be flushed to server when the table A is retrieved (**SELECT**)

- Some of dirty data can be flushed to server when a server function is called.

Transaction Termination and Restoration

Overview

The restore process in CUBRID makes it possible that the database is not affected even if a software or hardware error occurs. In CUBRID, all read and update commands that are made during a transaction must be atomic. This means that either all of the transaction's commands are committed to the database or none are. The concept of atomicity is extended to the set of operations that consists of a transaction. The transaction must either commit so that all effects are permanently applied to the database or roll back so that all effects are removed. To ensure transaction atomicity, CUBRID applies the effects of the committed transaction again every time an error occurs without the updates of the transaction being written to the disk. CUBRID also removes the effects of partially committed transactions in the database every time the site fails (some transactions may have not committed or applications may have requested to cancel transactions). This restore feature eases the burden for the applications of maintaining the database consistency depending on the system error. The restore process used in CUBRID is based on the undo/redo logging mechanism.

CUBRID provides an automatic restore method to maintain the transaction atomicity when a hardware or software error occurs. You do not have to take the responsibility for restore since CUBRID's restore feature always returns the database to a consistent state even when an application or computer system error occurs. For this purpose, CUBRID automatically rolls back part of committed transactions when the application fails or the user requests explicitly. For example, a system error that occurred during the execution of the **COMMIT WORK** statement must be stopped if the transaction has not committed yet (it cannot be confirmed that the user's operation has been committed). Automatic stop prevents errors causing undesired changes to the database by canceling uncommitted updates.

Restarting Database

CUBRID uses log volumes/files and database backups to restore committed or uncommitted transactions when a system or media (disk) error occurs. Logs are also used to support the user-specified rollback. A log consists of a collection of sequential files created by CUBRID. The most recent log is called the active log, and the rest are called archive logs. A log file refers to both the active log and archive logs.

All updates of the database are written to the log. Actually, two copies of the updates are logged. The first one is called a before image and used to restore data during execution of the user-specified **ROLLBACK WORK** statement or during media or system errors. The second copy is an after image and used to re-apply the updates when a media or system error occurs.

When the active log is full, CUBRID copies it to an archive log to store in the disk. The archive log is needed to restore the database when a system failure occurs. You don't need to maintain archive logs if there is no need for system failure restore. This configuration can be set by using the **media_failure_support** system parameter. For more information on this parameter, see Logging-Related Parameters.

Normal Termination or Error

CUBRID restores the database if it restarts due to a normal termination or a device error. The restore process re-applies the committed changes that have not been applied to the database and removes the uncommitted changes stored in the database. The general operation of the database resumes after the restore is completed. This restore process does not use any archive logs or database backup.

In a client/server environment, the database can restart by using server utilities.

Media Error

The user's intervention is somewhat needed to restart the database after a media error occurs. The first step is to restore the database by installing a backup of a known good state. In CUBRID, the most recent log file (the one after the last backup) must be installed. This specific log (archive or active) is applied to a backup copy of the database. As with normal termination, the database can restart after restoration is committed.

It is important to back up the database periodically. Backup periods differ depending on the frequency of database updates. Once a database backup is created, CUBRID uses the current database backup to specify the archive log that is not needed any more. However, CUBRID does not delete the archive log. The database administrator must take extra care when deleting the database backup or archive log. In some cases, the latest database backup may fail.

Note To minimize the possibility of losing database updates, it is recommended to create a snapshot of the archive log and backup the log to a disk before it is deleted from the disk. The DBA can backup and restore the database by using the **cubrid backupdb** and **cubrid restoredb** utilities. For more information on these utilities, see Database Backup.

Database User Authorization

Database User

CUBRID has two types of users by default: **DBA** and **PUBLIC**.

- All users have authorization granted to the **PUBLIC** user. All users of the database are automatically the members of **PUBLIC**. Granting authorization to the **PUBLIC** means granting it all users.
- The **DBA** user has the authorization of the database administrator. The **DBA** automatically becomes the member of all users and groups. That is, the **DBA** is granted the access for all tables. Therefore, there is no need to grant authorization explicitly to the **DBA** and **DBA** members. Each database user has a unique name. The database administrator can create multiple users simultaneously using the **cubrid createdb** utility (see How to Use the Database Management Utilities for details). A database user cannot have a member who already has the same authorization. If authorization is granted to a user, all members of the user is automatically granted the same authorization.

Managing User

Description

DBA and **DBA** members can create, drop and alter users by using SQL statements.

Syntax

```
CREATE USER user_name
[ PASSWORD password ]
[ GROUPS user_name [ {, user_name } ... ] ]
[ MEMBERS user_name [ {, user_name } ... ] ] ;
DROP USER user_name;
ALTER USER user_name PASSWORD password;
```

- *user_name* : Specifies the user name to create, delete or change.
- *password* : Specifies the user password to create or change.

Example 1

The following is an example in which the user Fred is created, the password is changed, and then the user Fred is deleted.

```
CREATE USER Fred;
ALTER USER Fred PASSWORD '1234';
DROP USER Fred;
```

Example 2

The following is an example in which a user is created and then members are added to the user. By the following statement, company becomes a group that has engineering, marketing and design as its members. marketing becomes a group with members smith and jones, design becomes a group with a member smith, and engineering becomes a group with a member brown.

```
CREATE USER company;
CREATE USER engineering GROUPS company;
CREATE USER marketing GROUPS company;
CREATE USER design GROUPS company;
CREATE USER smith GROUPS design, marketing;
CREATE USER jones GROUPS marketing;
CREATE USER brown GROUPS engineering;
```

Example 3

The following example creates the same groups as above, but uses the **MEMBERS** keyword instead of **GROUPS**.

```
CREATE USER smith;
```

```
CREATE USER brown;
CREATE USER jones;
CREATE USER engineering MEMBERS brown;
CREATE USER marketing MEMBERS smith, jones;
CREATE USER design MEMBERS smith;
CREATE USER company MEMBERS engineering, marketing, design;
```

Granting Authorization

Description

In CUBRID, the smallest grant unit of authorization is a table. You must grant appropriate authorization to other users (groups) before allowing them to access the table you created.

You don't need to grant authorization individually because the members of the granted group have the same authorization. The access to the (virtual) table created by a **PUBLIC** user is allowed to all other users. You can grant access authorization to a user by using the **GRANT** statement.

Syntax

```
GRANT operation [ { ,operation }_ ] ON table_name [ { ,table_name }_ ]
TO user [ { ,user }_ ] [ WITH GRANT OPTION ] [ ; ]
```

- *operation* : Indicates an operation that can be used when granting authorization. The following table shows the operations:
- **SELECT** : Allows to read the table definitions and retrieve records. The most general type of permissions.
- **INSERT** : Allows to create records in the table.
- **UPDATE** : Allows to modify the records already existing in the table.
- **DELETE** : Allows to delete records in the table.
- **ALTER** : Allows to modify the table definition, rename or delete the table.
- **INDEX** : Allows to call table methods or instance methods.
- **EXECUTE** : Allows to call table methods or instance methods.
- **ALL PRIVILEGES** : Includes all permissions described above.
- *table_name* : Specifies the name of the table or virtual table to be granted.
- *user* : Specifies the name of the user (group) to be granted. Enter the login name of the database user or **PUBLIC**, a system-defined user. If **PUBLIC** is specified, all database users are granted with the permission.
- **WITH GRANT OPTION** : **WITH GRANT OPTION** allows the grantee of authorization to grant that same privilege to another user.

Example 1

The following is an example in which the SELECT authorization for the olympic table is granted to Fred (all members of Fred).

```
GRANT SELECT ON olympic TO Fred;
```

Example 2

The following is an example in which **SELECT**, **INSERT**, **UPDATE** and **DELETE** authorization for the nation and athlete tables are granted to Jeniffer and Daniel (all members belonging to Jeniffer and Daniel).

```
GRANT SELECT, INSERT, UPDATE, DELETE ON nation, athlete TO  Jeniffer, Daniel;
```

Example 3

The following is an example in which all authorization for the game and event tables are granted to all users.

```
GRANT ALL PRIVILEGES ON game, event TO public;
```

Example 4

In the following example, the **GRANT** statement grants search authorization for the record and history tables to Ross, and **WITH GRANT OPTION** allows Ross to grant the same authorization to another user.

```
GRANT SELECT ON record, history TO Ross WITH GRANT OPTION;
```

Caution

- The grantor of authorization must be the owner of all tables listed before the grant operation or have **WITH GRANT OPTION** specified.
- Before granting **SELECT**, **UPDATE**, **DELETE** and **INSERT** authorization for a virtual table, the owner of the virtual table must have **SELECT** and GRANT authorization for all the tables included in the queries in the virtual table's query specification. The **DBA** user and the members of the **DBA** group are automatically granted all authorization for all tables.

Revoking Authorization

Description

You can revoke privileges using the **REVOKE** statement. The privileges granted to a user can be revoked anytime. If more than one privilege are granted to a user, all or part of the privileges can be revoked. In addition, if privileges on multiple tables are granted to more than one user using one **GRANT** statement, the privileges can be selectively revoked for specific users and tables.

If the privilege (**WITH GRANT OPTION**) is revoked from the grantor, the privilege granted to the grantee by that grantor is also revoked.

Syntax

```
REVOKE operation [ { , operation }_ ] ON table_name [ { , class_name }_ ]
FROM user [ { , user }_ ] [ ; ]
```

- *operation* : Indicates an operation that can be used when granting privileges. (See **Syntax** in Granting Privileges for details)
- *table_name* : Specifies the name of the table or virtual table to be granted.
- *user* : Specifies the name of the user (group) to be granted.

Example 1

The following is an example in which **SELECT**, **INSERT**, **UPDATE** and **DELETE** privileges for the nation and athlete tables are granted to Fred and John.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON nation, athlete TO Fred, John;
```

Example 2

The following is an example in which the **REVOKE** statement is used to allow John only the SELECT privilege while maintaining all the privileges for Fred granted in Example 1. If John granted the privileges to another user, the grantee is also allowed to use the SELECT privilege only.

```
REVOKE INSERT, UPDATE, DELETE ON nation, athlete FROM John;
```

Example 3

The following is an example in which the **REVOKE** statement is used to revoke all privileges granted to Fred in Example 1. If the statement is executed, Fred is not be allowed to perform any operation on the nation and athlete tables.

```
REVOKE ALL PRIVILEGES ON nation, athlete FROM Fred;
```

User Authorization Management METHOD

Description

The database administrator (**DBA**) can check and modify user authorization by calling authorization-related methods defined in **db_user** where information about database user is stored, and **db_authorizations** (the system authorization class). The administrator can specify **db_user** or **db_authorization** depending on the method to be called, and save the return value of a method to a variable. In addition, some methods can be called only by **DBA** or members of **DBA** group.

Syntax

```
CALL method_definition ON CLASS auth_class [ TO variable ] [ ; ]
CALL method_definition ON variable [ ; ]
```

login( ) method

As a class method of **db_user** class, this method is used to change the users who are currently connected to the database. The name and password of a new user to connect are given as parameters, and they must be string type. If there is no password, a blank character ('') can be used as the parameter. **DBA** and **DBA** members can call the **login( )** method without a password.

```
-- Connect as DBA user who has no password
CALL login ('dba', '') ON CLASS db_user;
-- Connect as a user_1 whose password is cubrid
CALL login ('user_1', 'cubrid') ON CLASS db_user;
```

add_user( ) method

As a class method of **db_user** class, this method is used to add a new user. The name and password of a new user to add are given as parameters, and they must be string type. At this time, the new user name should not duplicate any user name already registered in a database. The **add_user( )** can be called only by **DBA** or members of **DBA** group.

```
-- Add user_2 who has no password
CALL add_user ('user_2', '') ON CLASS db_user;
-- Add user_3 who has no password, and save the return value of a method into an admin
variable
CALL add_user ('user_2', '') ON CLASS db_user to admin;
```

drop_user( ) method

As a class method of **db_user** class, this method is used to drop an existing user. Only the user name to be dropped is given as a parameter, and it must be a string type. However, the owner of a class cannot be dropped thus **DBA** needs to specify a new owner of the class before dropping the user. The **drop_user( )** method can be also called only by **DBA** or members of **DBA**.

```
-- Delete user_2
CALL drop_user ('user_2') ON CLASS db_user;
```

find_user( ) method

As a class method of **db_user** class, this method is used to find a user who is given as a parameter. The name of a user to be found is given as a parameter, and the return value of the method is stored into a variable that follows 'to'. The stored value can be used in a next query execution.

```
-- Find user_2 and save it into a variable called 'admin'
CALL find_user ('user_2') ON CLASS db_user to admin;
```

set_password( ) method

This method is an instance method that can call each user instance, and it is used to change a user's password. The new password of a specified user is given as a parameter. General users other than **DBA** and **DBA** group members can only change their own passwords.

```
-- Add user_4 and save it into a variable called user_common
CALL add_user ('user_4','') ON CLASS db_user to user_common;
```

```
-- Change the password of user_4 to 'abcdef'
CALL set_password('abcdef') on user_common;
```

change_owner() method

As a class method of **db_authorizations** class, this method is used to change the owner of a class. The name of a class for which you want to change the owner, and the name of a new owner are given as parameters. At this time, the class and owner that are specified as a parameter must exist in a database. Otherwise, an error occurs. **change_owner( )** can be called only by **DBA** or members of **DBA** group.

```
-- Change the owner of table_1 to user_4
CALL change_owner ('table_1', 'user_4') ON CLASS db_authorizations;
```

Example

The following is an example of a **CALL** statement that calls the find_user method defined in the system table **db_user**. It is called to determine whether the database user entered as the **find_user** exists. The first statement calls the table method defined in the **db_user** class. The name (**db_user** in this case) is stored in x if the user is registered in the database. Otherwise, **NULL** is stored.

The second statement outputs the value stored in the variable x. In this query statement, the **DB_ROOT** is a system class that can have only one record. It can be used to output the value of sys_date or other registered variables. For this purpose, the **DB_ROOT** can be replaced by another table having only one record.

```
CALL find_user('dba') ON CLASS db_user to x;
Result
======================
db_user

SELECT x FROM db_root;
x
======================
db_user
```

With **find_user**, you can determine if the user exists in the database depending on whether the returned value is **NULL** or not.

Query Optimization

Updating Statistics

Description

With the **UPDATE STATISTICS ON** statement, you can generate internal statistics used by the query processor. Such statistics allow the database system to perform query optimization more efficiently.

Syntax

```
UPDATE STATISTICS ON { table_spec [ {, table_spec } ] | ALL CLASSES | CATALOG CLASSES }
[ ; ]
table_spec :
single_table_spec
( single_table_spec [ {, single_table_spec } ] )
single_table_spec :
[ ONLY ] table_name
| ALL table_name [ ( EXCEPT table_name ) ]
```

- **ALL CLASSES** : If the **ALL CLASSES** keyword is specified, the statistics on all the tables existing in the database are updated.

Checking Statistics Information

Description

You can check the statistics Information with the session command of the CSQL interpreter.

Syntax

```
csql> ;info stats <table_name>
```

- *table_name* : Table name to check the statistics Information

Example

The following is an example that outputs the statistics Information of the t1 table in the CSQL interpreter.

```
CREATE TABLE t1 (code INT);
INSERT INTO t1 VALUES(1),(2),(3),(4),(5); CREATE INDEX ON t1(code); UPDATE STATISTICS ON
t1;
;info stats t1
CLASS STATISTICS ****************  Class name: t1 Timestamp: Mon Mar 14 16:26:40
2011  Total pages in class heap: 1  Total objects: 5  Number of attributes: 1  Atrribute:
code     id: 0     Type: DB_TYPE_INTEGER     Mininum value: 1     Maxinum value:
5    B+tree statistics:        BTID: { 0 , 1049 }        Cardinality: 5 (5) , Total
pages: 2 , Leaf pages: 1 , Height: 2
```

Using SQL Hint

Description

Using hints can affect the performance of query execution. you can allow the query optimizer to create more efficient execution plan by referring the SQL HINT. The SQL HINTs related tale join, index, and statistics information are provided by CUBRID.

Syntax

```
CREATE /*+ NO_STATS */ [TABLE | CLASS] ...;
ALTER /*+ NO_STATS */ [TABLE | CLASS] ...;
```

```
CREATE /*+ NO_STATS */ INDEX ...;
ALTER /*+ NO_STATS */ INDEX ...;
DROP /*+ NO_STATS */ INDEX ...;

SELECT /*+ hint [ { hint } ... ] */
SELECT --+ hint [ { hint } ... ]
SELECT //+ hint [ { hint } ... ]

hint :
USE_NL[(spec-name[{, spec-name}...])]
USE_IDX[(spec-name[{, spec-name}...])]
USE_MERGE[(spec-name[{, spec-name}...])]
ORDERED
QUERY_CACHE(1)
USE_DESC_IDX
NO_COVERING_IDX
```

SQL hints are specified by using plus signs and comments. CUBRID interprets this comment as a list of hints separated by blanks. The hint comment must appear after the **SELECT**, **CREATE**, or **ALTER** keyword, and the comment must begin with a plus sign (+), following the comment delimiter.

- *hint* : The following hints can be specified.
- **USE_NL** : Related to a table join, the query optimizer creates a nested loop join execution plan with this hint.
- **USE_MERGE** : Related to a table join, the query optimizer creates a sort merge join execution plan with this hint.
- **ORDERED** : Related to a table join, the query optimizer create a join execution plan with this hint, based on the order of tables specified in the FROM clause. The left table in the FROM clause becomes the outer table; the right one becomes the inner table.
- **USE_IDX** : Related to a index, the query optimizer creates a index join execution plan corresponding to a specified table with this hint.
  - **USE_DESC_IDX** : This is a hint for the scan in descending order. For more information, see Index Scan in Descending Order.
  - **NO_COVERING_IDX** : This is a hint not to use the covering index feature. For more information, see Covering Index.
- **NO_STATS** : Related to statistics information, the query optimizer does not update statistics information. Query performance for the corresponding queries can be improved; however, query plan is not optimized because the information is not updated.
- *spec_name* : If the *spec_name* is specified together with **USE_NL**, **USE_IDX** or **USE_MERGE**, the specified join method applies only to the *spec_name*. If **USE_NL** and **USE_MERGE** are specified together, the given hint is ignored. In some cases, the query optimizer cannot create a query execution plan based on the given hint. For example, if **USE_NL** is specified for a right outer join, the query is converted to a left outer join internally, and the join order may not be guaranteed.
- **QUERY_CACHE(1)** : The query cache feature will be applied only to the queries to which this hint is given in the second query cache mode. If you set 2 for the system parameter, **query_cache_mode**, you can call it the second query cache mode. For more information, see Query/Cache-Related Parameters.

Example 1

The following is an example of retrieving the years when Sim Kwon Ho won medals and the types of medals. Here, a nested loop join execution plan needs to be created which has the **athlete** table as an outer table and the **game** table as an inner table. It can be expressed by the following query. The query optimizer creates a nested loop join execution plan that has the **game** table as an outer table and the **athlete** table as an inner table.

```
SELECT /*+ USE_NL ORDERED  */ a.name, b.host_year, b.medal
FROM athlete a, game b WHERE a.name = 'Sim Kwon Ho' AND a.code = b.athlete_code;
  name                      host_year   medal
=========================================================
  'Sim Kwon Ho'                  2000   'G'
  'Sim Kwon Ho'                  1996   'G'
2 rows selected.
```

Example 2

The following is an example of viewing query execution time with **NO_STAT** hint to improve the functionality of drop partitioned table (before_2008); any data is not stored in the table. Assuming that there are more than 1 million data in the **participant2** table. The execution time in the example can differ depending on system performance and database configuration.

```
-- Not using NO_STATS hint
ALTER TABLE participant2 DROP partition before_2008;
SQL statement execution time: 31.684550 sec
Current transaction has been committed.
1 command(s) successfully processed.

-- Using NO_STATS hint
ALTER /*+ NO_STATS */ TABLE participant2 DROP partition before_2008;
SQL statement execution time: 0.025773 sec
Current transaction has been committed.
1 command(s) successfully processed.
```

Viewing Query Plan

Description

To view a query plan for a CUBRID SQL query, change the value of the optimization level by using the **SET OPTIMIZATION** statement. You can get the current optimization level value by using the **GET OPTIMIZATION** statement.

The CUBRID query optimizer determines whether to perform query optimization and output the query plan by referencing the optimization level value set by the user. The query plan is displayed as standard output; the following explanations are based on the assumption that the plan is used in a terminal-based program such as the CSQL Interpreter. In the CSQL query editor, you can view execution plan by executing the **;plan** command. See Session Commands. For information on how to view a query plan, see the CUBRID Manager.

Syntax

```
SET OPTIMIZATION LEVEL opt-level [;]
GET OPTIMIZATION LEVEL [ { TO | INTO } variable ] [;]
```

- *opt-level*: A value that specifies the optimization level. It has the following meanings.
- 0 : Does not perform query optimization. The query is executed using the simplest query plan. This value is used only for debugging.
- 1 : Create a query plan by performing query optimization and executes the query. This is a default value used in CUBRID, and does not have to be changed in most cases.
- 2 : Creates a query plan by performing query optimization. However, the query itself is not executed. Generally, this value is not used; it is used together with the following values to be set for viewing query plans.
- 257 : Performs query optimization and outputs the created query plan. This value works for displaying the query plan by internally interpreting the value as 256+1 related with the value 1.
- 258 : Performs query optimization and outputs the created query plan. The difference from the value 257 is that the query is not executed. That is, this value works for displaying the query plan by internally interpreting the value as 256+2 related with the value 2. This setting is useful to examine the query plan but not to intend to see the query results.
- 513 : Performs query optimization and outputs the detailed query plan. This value works for displaying more detailed query plan than the value 257 by internally interpreting the value as 512+1.
- 514 : Performs query optimization and outputs the detailed query plan. However, the query is not executed. This value works for displaying more detailed query plan than the value 258 by internally interpreting the value as 512+2.

Example

The following example is to display the query plan but not execute a query itself by setting the optimization level to 258, the query is that retrieves the years when Sim Kwon Ho won medals and the types of medals.

```
GET OPTIMIZATION LEVEL
              Result
=============
                        1
 SET OPTIMIZATION LEVEL 258;
```

```
SELECT a.name, b.host_year, b.medal
FROM athlete a, game b WHERE a.name = 'Sim Kwon Ho' AND a.code = b.athlete_code
Query plan:
  Nested loops
        Sequential scan(game b)
        Index scan(athlete a, pk_athlete_code, a.code=b.athlete_code)
There are no results.
0 rows selected.
```

Using Indexes

USING INDEX Clause

Description

The **USING INDEX** clause forces a sequential scan or an index scan to be used or an index for better performance to be included. The **USING INDEX** clause must be specified after the **WHERE** clause of **SELECT**, **DELETE**, or **UPDATE** statement.

If you specify the list of index names in the **USING INDEX** clause, the query optimizer calculates the query execution cost and makes the most optimized execution plan comparing the cost between the index scan and the sequential scan which are specified (CUBRID performs the query optimization based on the cost to select the execution plan).

You can use the **USING INDEX** clause in the order that you want without using **ORDER BY**. If you do an index scan in CUBRID, the result will be created in the order of being saved in the index and you can **USING INDEX** to get the query result in the specific index order when one table has multiple indexes.

Syntax

```
SELECT . . . FROM . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ...] } ] [ ; ]
DELETE FROM . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ...] } ] [ ; ]
UPDATE . . . SET . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ...] } ] [ ; ]
index_spec :
 [table_name.]index_name [(+)]
```

- **NONE** : If **NONE** is specified, a sequential scan is selected.
- (+) : If (+) is specified after the index name, an index scan using the specified index is selected.

Example

The following is an example of creating an index based on the table creation statement of the **athlete** table.

```
CREATE TABLE athlete (
   code             SMALLINT    NOT NULL PRIMARY KEY,
   name             VARCHAR(40) NOT NULL,
   gender           CHAR(1)     ,
   nation_code      CHAR(3)     ,
   event            VARCHAR(30)
   );
CREATE UNIQUE INDEX athlete_idx ON athlete(code, nation_code);
CREATE INDEX char_idx ON athlete(gender, nation_code);
```

For the following query, the query optimizer can choose an index scan that uses the **athlete_idx** index.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA';
```

As in the query below, if **USING INDEX char_idx** is specified, the query optimizer calculates the index scan cost only for the given index specified by **USING INDEX**.

If the index scan cost is less than the sequential scan cost, an index scan is performed.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx;
```

To forcefully specify an index scan that uses the **char_idx** index, place (+) after the index name.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx(+);
```

To allow a sequential scan to be selected, specify **NONE** in the **USING INDEX** clause as follows:

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX NONE;
```

If more than one indexes were specified in the **USING INDEX** clause as shown below, the query optimizer chooses an appropriate one from the specified indexes.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx, athlete_idx;
```

If you execute queries for multiple tables, you can specify to perform an index scan on one table by using a special index, and a sequential scan on other tables. These queries have the following form.

```
SELECT ... FROM tab1, tab2 WHERE ... USING INDEX tab1.idx1, tab2.NONE;
```

If you execute a query including the **USING INDEX** clause, the query optimizer considers all indexes available of the corresponding table for the tables not specified indexes. For example, if the table tab1 has indices idx1 and idx2, and the table tab2 has indices idx3, idx4 and idx5, specify the index for only tab1 and if if you do not specify tab2 index, the query optimizer works considering tab2 index.

```
SELECT ... FROM tab1, tab2 WHERE ... USING INDEX tab1.idx1;
```

- Select the best query plan by comparing the sequential scan and index scan of table tab1.
- Select the most optimized query plan by comparing the sequential scan on the table tab2 and the index scan on idx3, idx4 and idx5.

To perform an index scan on the table tab2 and a sequential scan on the table tab1, specify tab1.NONE so as not to perform an index scan on the tab1 table.

```
SELECT * from tab1,tab2 WHERE tab1.id > 2 and tab2.id < 3 USING index i_tab2_id, tab1.NONE;
```

Index Scan in Descending Order

Description

When a query is executed by sorting in descending order as follows, it usually creates a reverse index.

```
SELECT * FROM tab [WHERE ...] ORDER BY a DESC
```

However, if you create an ascending index and an descending index in the same column, the possibility of deadlock increases. In order to decrease the possibility of such case, CUBRID supports the descending scan without the separate descending index creation. Users can use the **USE_DESC_IDX** hint to specify the use of the descending scan. If the hint is not specified, the following three query executions should be considered, provided that the columns listed in the **ORDER BY** clause can use the index.

- Sequential scan + Sort in descending order
- Scan in general ascending order + sort in descending
- Scan in descending order that does not require a separate scan

Although the **USE_DESC_IDX** hint is omitted for the scan in descending order, the query optimizer decides the last execution plan of the three listed for an optimal plan.

Note The **USE_DESC_IDX** hint is not supported for the join query.

Example

```
CREATE TABLE di (i INT);
CREATE INDEX on di (i);
INSERT INTO di VALUES (5),(3),(1),(4),(3),(5),(2),(5);
```

The following is an example of executing a query with the USE_DESC_IDX hint.

```
-- We now run the following query, using the ''use_desc_idx'' SQL hint:

SELECT /*+ USE_DESC_IDX */ * FROM di WHERE i > 0 LIMIT 3;
```

```
Query plan:
 Index scan(di di, i_di_i, (di.i range (0 gt_inf max) and inst_num() range (min inf_le 3))
(covers) (desc_index))

            i
=============
            5
            5
            5
```

The following example shows that if the query doesn't have the USE_DESC_IDX even though it is same as the above, it can not be scanned in descending order, and the output result may be different.

```
-- The same query, without the hint, will have a different output, since descending scan
is not used.

SELECT  * FROM di WHERE i > 0 LIMIT 3;

Query plan:

Index scan(di di, i_di_i, (di.i range (0 gt_inf max) and inst_num() range (min inf_le 3))
(covers))

            i
=============
            1
            1
            1
```

The following example has the same query and requests the sorting in descending order with ORDER BY DESC. There is no USE_DESC_IDX hint in this case but the output result is the same as in the first example, because it is scanned in descending order.

```
-- We also run the same query , this time asking that the results are displayed in
descending order. However, no hint will be given. Since the

-- ORDER BY...DESC clause is present, CUBRID will use descending scan, even if the hint
is  was not given, thus avoiding to sort the records.

SELECT * FROM di WHERE i > 0 ORDER BY i DESC LIMIT 3;

Query plan:
 Index scan(di di, i_di_i, (di.i range (0 gt_inf max)) (covers) (desc_index))

            i
=============
            5
            5
            5
```

Covering Index

Description

The covering index is the index including the data of all columns in the **SELECT** list and the **WHERE**, **HAVING**, **GROUP BY**, and **ORDER BY** clauses.

You only need to scan the index pages, as the covering index contains all the data necessary for executing a query, and it also reduces the I/O costs as it is not necessary to scan the data storage any further. To increase data search speed, you can consider creating a covering index but you should be aware that the **INSERT** and the **DELETE** processes may be slowed down due to the increase in index size.

The rules about the applicability of the covering index are as follows:

- If the covering index is applicable, you should use the CUBRID query optimizer first.
- For the join query, if the index includes columns of the table in the **SELECT** list, use this index.
- You can not use the covering index if an index can not be used.

Example

```
CREATE TABLE t (col1 INT, col2 INT, col3 INT);
CREATE INDEX ON t (col1,col2,col3);
INSERT INTO t VALUES (1,2,3),(4,5,6),(10,8,9);
```

The following example shows that the index is used as a covering index because both the column **SELECT** and the column with **WHERE** condition exist within the index.

```
SELECT * FROM t WHERE col1 < col3;

Query plan:
 Index scan(t t, i_t_col1_col2_col3, [(t.col1 range (min inf_lt t.col3))] (covers))
         col1         col2         col3
=======================================
            1            2            3
            4            5            6
```

Caution

If the covering index is applied when you get the values from the **VARCHAR** type column, the empty strings that follow will be truncated. If the covering index is applied to the execution of query optimization, the resulting query value will be retrieved. This is because the value will be stored in the index with the empty string being truncated.

If you don't want this, use the **NO_COVERING_IDX** hint, which does not use the covering index function. If you use the hint, you can get the result value from the data area rather than from the index area.

The following is a detailed example of the above situation. First, create a table with columns in **VARCHAR** types, and then **INSERT** the value with the same start character string value but the number of empty characters. Next, create an index in the column.

```
CREATE TABLE tab(c VARCHAR(32));
INSERT INTO tab VALUES('abcd'),('abcd    '),('abcd ');
CREATE INDEX ON tab(c);
```

If you must use the index (the covering index applied), the query result is as follows:

```
SELECT * FROM tab where c='abcd    ' USING INDEX i_tab_c(+);
 c
======================
'abcd'
'abcd'
'abcd'
```

The following is the query result when you don't use the index.

```
SELECT * FROM tab WHERE c='abcd    ' USING INDEX tab.NONE;
 c
======================
'abcd'
'abcd    '
'abcd '
```

As you can see in the above comparison result, the value in the **VARCHAR** type retrieved from the index will appear with the following empty string truncated when the covering index has been applied.

Ordered Index

Description

The index including all columns in the **ORDER BY** clause is referred to as the ordered index. In general, for an ordered index, the columns in the **ORDER BY** clause should be located at the front of the index.

```
SELECT * FROM tab WHERE col1 > 0 ORDER BY col1, col2
```

- The index consisting of tab(col1, col2) is an ordered index.
- The index consisting of tab(col1, col2, col3) is also an ordered index. This is because the col3, which is not referred by the ORDER BY clause comes after col1 and col2.
- The index consisting of tab(col1) is not an ordered index.

- You can use the index consisting of tab(col3, col1, col2) or tab(col1, col3, col2) for optimization. This is because col3 is not located at the back of the columns in the ORDER BY clause.

Although the columns composing an index do not exist in the **ORDER BY** clause, you can use an ordered index if the column condition is a constant.

```
SELECT * FROM tab WHERE col2=val ORDER BY col1,col3;
```

If the index consisting of tab(col1, col2, col3) exists and the index consisting of tab(col1, col2) do not exist when executing the above query, the query optimizer uses the index consisting of tab(col1, col2, col3) as an ordered index. You can get the result in the requested order when you execute an index scan, so you don't need to sort rows.

If you can use the sorted index and the covering index, use the latter first. If you use the covering index, you don't need to retrieve additional data, because the data result requested is included in the index page, and you won't need to sort the result if you are satisfied with the index order.

If the query doesn't include any conditions and uses an ordered index, the ordered index will be used under the condition that the first column meets the **NOT NULL** condition.

Example

```
CREATE TABLE tab (i INT, j INT, k INT);
CREATE INDEX on tab (j,k);
INSERT INTO tab VALUES (1,2,3),(6,4,2),(3,4,1),(5,2,1),(1,5,5),(2,6,6),(3,5,4);
```

The following example shows that the j and k columns execute **ORDER BY** so that the index consisting of tab(j,k) will be the ordered index and will not go through a separate sorting process.

```
SELECT i,j,k FROM tab WHERE j > 0 ORDER BY j,k;

--  the  selection from the query plan dump shows that the ordering index i_tab_j_k was
used and sorting was not necessary
--  (/* --> skip ORDER BY */)
Query plan:
iscan
    class: tab node[0]
    index: i_tab_j_k term[0]
    sort:  2 asc, 3 asc
    cost:  fixed 0(0.0/0.0) var 1(0.0/1.0) card 0
Query stmt:
select tab.i, tab.j, tab.k from tab tab where ((tab.j> ?:0 )) order by 2, 3
/* ---> skip ORDER BY */

            i            j            k
=======================================
            5            2            1
            1            2            3
            3            4            1
            6            4            2
            3            5            4
            1            5            5
            2            6            6
```

The following example shows that the j and k columns execute **ORDER BY** and the index including all columns that **SELECT**, so that the index consisting of tab(j,k) will be the covering index. Therefore, the value will be retrieved from the index itself and it will not go through a separate sorting process.

```
SELECT /*+ RECOMPILE */ j,k FROM tab WHERE j > 0 ORDER BY j,k;

--  in this case the index i_tab_j_k is a covering index and also respects the orderind
index property.
--  Therefore, it is used as a covering index and sorting is not performed.

Query plan:
iscan
    class: tab node[0]
    index: i_tab_j_k term[0] (covers)
    sort:  1 asc, 2 asc
    cost:  fixed 0(0.0/0.0) var 1(0.0/1.0) card 0

Query stmt: select tab.j, tab.k from tab tab where ((tab.j> ?:0 )) order by 1, 2
```

```
/* ---> skip ORDER BY */

            j            k
==========================
            2            1
            2            3
            4            1
            4            2
            5            4
            5            5
            6            6
```

The following example shows that when there is i column condition, the **ORDER BY** will be executed to j and k column and the columns to **SELECT** are i, j and k, the index consisting of tab(i,j,k) will be used as the covering index. The value is retrieved from the index itself but it will go through a separate sorting process for the ORDER BY j, k.

```
CREATE INDEX ON tab (i,j,k);
SELECT /*+ RECOMPILE */ i,j,k FROM tab WHERE i > 0 ORDER BY j,k;

-- since an index on (i,j,k) is now available, it will be used as covering index. However, sorting the results according to
-- the ORDER BY  clause is needed.
Query plan:
temp(order by)
    subplan: iscan
                 class: tab node[0]
                 index: i_tab_i_j_k term[0] (covers)
                 sort:  1 asc, 2 asc, 3 asc
                 cost:  fixed 0(0.0/0.0) var 1(0.0/1.0) card 1
    sort:  2 asc, 3 asc
    cost:  fixed 6(5.0/1.0) var 1(0.0/1.0) card 1

Query stmt: select tab.i, tab.j, tab.k from tab tab where ((tab.i> ?:0 )) order by 2, 3

            i            j            k
=======================================
            5            2            1
            1            2            3
            3            4            1
            6            4            2
            3            5            4
            1            5            5
            2            6            6
```

GROUP BY Clause Optimization

Description

GROUP BY caluse optimization works on the premise that if all columns in the **GROUP BY** clause are included in an index, you can use the index upon executing a query, so you don't execute a separate sorting job. The columns in the **GROUP BY** clause must exist in front side of the column forming the index.

```
SELECT * FROM tab WHERE col1 > 0 GROUP BY col1,col2
```

- You can use the index consisting of tab(col1, col2) for optimization.
- The index consisting of tab(col1, col2, col3) can be used because col3 no referred by **GROUP BY** comes after col1 and col2.
- You can not use the index consisting of tab(col1) for optimization.
- You also can not use the index consisting of tab(col3, col1, col2) or tab(col1, col3, col2), because col3 is not located at the back of the column in the **GROUP BY** clause.

You can use the index if the column condition is a constant although the column consisting of the idex doesn't exist in the **GROUP BY** clause.

```
SELECT * FROM tab WHERE col2=val GROUP BY col1,col3
```

If there is any index that consists of tab(col1, col2, col3) in the above example, use the index for optimizing **GROUP BY**.

Row sorting by **GROUP BY** is not required, because you can get the result as the requested order on the index scan.

If the index consisting of the **GROUP BY** column and the first column of the index is **NOT NULL**, even though there is no **WHERE** clause, the **GROUP BY** optimization will be applied.

GROUP BY optimization is applied only when **MIN**() or **MAX**() are used in an aggregate function, and to use the two aggregate functions together, an identical column must be used.

```
CREATE INDEX ON T(a, b, c);
SELECT a, MIN(b), c, MAX(b) FROM T WHERE a > 18 GROUP BY a, b;
```

Example

```
CREATE TABLE tab (i INT, j INT, k INT);
CREATE INDEX ON tab (j,k);
INSERT INTO tab VALUES (1,2,3),(6,4,2),(3,4,1),(5,2,1),(1,5,5),(2,6,6),(3,5,4);
```

The following example executes the **GROUP BY** to j and k column so that the index can be used without a separate sorting process.

```
SELECT i,j,k FROM tab WHERE j > 0 GROUP BY j,k;

--  the  selection from the query plan dump shows that the index i_tab_j_k was used and
sorting was not necessary
--  (/* ---> skip GROUP BY */)

Query plan:
iscan
    class: tab node[0]
    index: i_tab_j_k term[0]
    sort:  2 asc, 3 asc
    cost:  fixed 0(0.0/0.0) var 1(0.0/1.0) card 0

Query stmt:
select tab.i, tab.j, tab.k from tab tab where ((tab.j> ?:0 )) group by tab.j, tab.k
/* ---> skip GROUP BY */
            i            j            k
            5            2            1
            1            2            3
            3            4            1
            6            4            2
            3            5            4
            1            5            5
           2            6            6
```

The following example executes **GROUP BY** to the j and k column and there is no condition for the j but the j column has the **NOT NULL** attribute so that, the index consisting of tab(j,k) will be used without a separate sorting process.

```
ALTER TABLE tab CHANGE COLUMN j j INT NOT NULL;
SELECT * FROM tab GROUP BY j,k;

--  the  selection from the query plan dump shows that the index i_tab_j_k was used (since
j has the NOT NULL constraint )
--  and sorting was not necessary (/* ---> skip GROUP BY */)
Query plan:
iscan
    class: tab node[0]
    index: i_tab_j_k
    sort:  2 asc, 3 asc
    cost:  fixed 0(0.0/0.0) var 1(0.0/1.0) card 0

Query stmt: select tab.i, tab.j, tab.k from tab tab group by tab.j, tab.k
/* ---> skip GROUP BY */
=== <Result of SELECT Command in Line 1> ===
            i            j            k
=======================================
            5            2            1
            1            2            3
            3            4            1
            6            4            2
            3            5            4
            1            5            5
            2            6            6
```

TRIGGER

CREATE TRIGGER

Guideline for TRIGGER Definition

Trigger definition provides various and powerful functionalities. Before creating a trigger, you must consider the following:

- **Does the trigger condition expression cause unexpected results (side effects)?**

  You must use the SQL statements within an expectable range.

- **Does the trigger action change the table given as its event target?**

  While this type of design is not forbidden in the trigger definition, it must be carefully applied, because a trigger can be created that falls into an infinite loop. When the trigger action modifies the event target table, the same trigger can be called again. If a trigger occurs in a statement that contains a **WHERE** clause, there is no side effect in the table affected by the **WHERE** clause.

- **Does the trigger cause unnecessary overhead?**

  If the desired action can be expressed more effectively in the source, implement it directly in the source.

- **Is the trigger executed recursively?**

  If the trigger action calls a trigger and this trigger calls the previous trigger again, a recursive loop is created in the database. If a recursive loop is created, the trigger may not be executed correctly, or the current session must be forced to terminate to break the ongoing infinite loop.

- **Is the trigger definition unique?**

  A trigger defined in the same table or the one started in the same action becomes the cause of an unrecoverable error. A trigger in the same table must have a different trigger event. In addition, trigger priority must be explicitly and unambiguously defined.

TRIGGER Definition

Description

A trigger is created by defining a trigger target, condition and action to be performed in the **CREATE TRIGGER** statement. A trigger is a database object that performs a defined action when a specific event occurs in the target table.

Syntax

```
CREATE TRIGGER trigger_name
[ STATUS { ACTIVE | INACTIVE } ]
[ PRIORITYkey ]
event_time event_type[ event_target ]
[ IFcondition ]
EXECUTE [ AFTER | DEFERRED ] action [ ; ]

event_time:
   • BEFORE
   • AFTER
   • DEFERRED

event_type:
   • INSERT
   • STATEMENT INSERT
   • UPDATE
   • STATEMENT UPDATE
   • DELETE
   • STATEMENT DELETE
   • ROLLBACK
   • COMMIT

event_target:
```

```
    • ONtable_name
    • ONtable_name [ (column_name) ]

condition:
    • expression

action:
  • REJECT
  • INVALIDATE TRANSACTION
  •  PRINT message_string
  •  INSERT statement
  •  UPDATE statement
  •  DELETE statement
```

- *trigger_name* : Specifies the name of the trigger to be defined.
- [**STATUS** { **ACTIVE** | **INACTIVE** }] : Defines the state of the trigger (if not defined, the default value is **ACTIVE**).
- If **ACTIVE** state is specified, the trigger is executed every time the corresponding event occurs.
- If **INACTIVE** state is specified, the trigger is not executed even when the corresponding event occurs. The state of the trigger can be modified. For more information, see Altering TRIGGER Definition section.
- [**PRIORITY** *key*] : Specifies a trigger priority if multiple triggers are called for an event. *key* must be a floating point value that is not negative. If the priority is not defined, the lowest priority 0 is assigned. Triggers having the same priority are executed in a random order. The priority of triggers can be modified. For more information, see Altering TRIGGER Definition section.
- *event_time* : Specifies the point of time when the conditions and actions are executed. **BEFORE**, **AFTER** or **DEFERRED** can be specified. For more information, see the Event Time section.
- *event_type* : Trigger types are divided into a user trigger and a table trigger. For more information, see the TRIGGER Event Type section.
- *event_target* : An event target is used to specify the target for the trigger to be called. For more information, see the TRIGGER Event Target section.
- *condition* : Specifies the trigger condition. For more information, see the TRIGGER Condition section.
- *action* : Specifies the trigger action. For more information, see the TRIGGER Action section.

Example

The following is an example of creating a trigger that rejects the update if the number of medals won is smaller than 0 when an instance of the participant table is updated.

As shown below, the update is rejected if you try to change the number of gold medals that Korea won in the 2004 Olympic Games to a negative number.

```
CREATE TRIGGER medal_trigger
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;

UPDATE participant SET gold = -5 WHERE nation_code = 'KOR'
AND host_year = 2004;

ERROR: The operation has been rejected by trigger "medal_trigger".
```

Event Time

Description

Specifies the point of time when trigger conditions and actions are executed. The types of event time are **BEFORE**, **AFTER** and **DEFERRED**.

- **BEFORE** : Checks the condition before the event is processed.
- **AFTER** : Checks the condition after the event is processed.
- **DEFERRED** : Checks the condition at the end of the transaction for the event. If you specify **DEFERRED**, you cannot use **COMMIT** or **ROLLBACK** as the event type.

Trigger Type

User Trigger

- A trigger relevant to a specific user of the database is called a user trigger.
- A user trigger has no event target and is executed only by the owner of the trigger (the user who created the trigger).
- Event types that define a user trigger are **COMMIT** and **ROLLBACK**.

Table Trigger

- A trigger that has a table as the event target is called a table trigger (class trigger).
- A table trigger can be seen by all users who have the **SELECT** privilege on the target table.
- Event types that define a table trigger are instance and statement events.

TRIGGER Event Type

Description

- Instance events : An event type whose unit of operation is an instance. The types of instance events are as follows:
- **INSERT**
- **UPDATE**
- **DELETE**
- Statement events : If you define a statement event as an event type, the trigger is called only once when the trigger starts even when there are multiple objects (instances) affected by the given statement (event). The types of statement events are as follows:
- **STATEMENT INSERT**
- **STATEMENT UPDATE**
- **STATEMENT DELETE**
- Other events : **COMMIT** and **ROLLBACK** cannot be applied to individual instances.
- **COMMIT**
- **ROLLBACK**

Example 1

The following is an example of using an instance event. The example trigger is called by each instance affected by the database update. For example, if the score values of five instances in the history table are modified, the trigger is called five times. If you want the trigger to be called only once, before the first instance of the score column is updated, use the **STATEMENT UPDATE** type as in example 2.

```
CREATE TRIGGER example
...
BEFORE UPDATE ON history(score)
...
```

Example 2

The following is an example of using a statement event. If you define a statement event, the trigger is called only once before the first instance gets updated even when there are multiple instances affected by the update.

```
CREATE TRIGGER example
...
BEFORE STATEMENT UPDATE ON history(score)
...
```

Caution

- You must specify the event target when you define an instance or statement event as the event type.
- **COMMIT** and **ROLLBACK** cannot have an event target.

TRIGGER Event Target

Description

An event target specifies the target for the trigger to be called. The target of a trigger event can be specified as a table or column name. If a column name is specified, the trigger is called only when the specified column is affected by the event. If a column is not specified, the trigger is called when any column of the table is affected. Only **UPDATE** and **STATEMENT UPDATE** events can specify a column as the event target.

Example

The following is an example of specifying the score column of the history table as the event target of the example trigger.

```
CREATE TRIGGER example
...
BEFORE UPDATE ON history(score)
...
```

Combination of Event Type and Target

Description

A database event calling triggers is identified by the trigger event type and event target in a trigger definition. The following table shows the trigger event type and target combinations, along with the meaning of the CUBRID database event that the trigger event represents.

Event Type	Event Target	Corresponding Database Activity
UPDATE	Table	Trigger is called whenever any column of the table is updated.
STATEMENT UPDATE	Table	Trigger is called whenever an **UPDATE** statement is executed on the table.
INSERT	Table	Trigger is called whenever an instance of the table is created.
STATEMENT INSERT	Table	Trigger is called whenever an **INSERT** statement is executed on the table.
DELETE	Table	Trigger is called whenever an instance of the table is deleted.
STATEMENT DELETE	Table	Trigger is called whenever a **DELETE** statement is executed on the table.
COMMIT	None	Trigger is called whenever a database transaction is committed. The **COMMIT WORK** statement initiates this trigger.
ROLLBACK	None	Trigger is called whenever the database transaction is rolled back. The **ROLLBACK WORK** statement initiates this trigger.

TRIGGER Condition

Description

You can specify whether a trigger action is to be performed by defining a condition when defining the trigger.

- If a trigger condition is specified, it can be written as an independent compound expression that evaluates to true or false. In this case, the expression can contain arithmetic and logical operators allowed in the **WHERE** clause of the **SELECT** statement. The trigger action is performed if the condition is true; if it is false, action is ignored.
- If a trigger condition is omitted, the trigger becomes an unconditional trigger, which refers to that the trigger action is performed whenever it is called.

Example 1

The following is an example of using a correlation name in an expression within a condition. If the event type is **INSERT**, **UPDATE** or **DELETE**, the expression in the condition can reference the correlation names **obj**, **new** or **old** to access a specific column. This example prefixes **obj** to the column name in the trigger condition to show that the example trigger tests the condition based on the current value of the record column.

```
CREATE TRIGGER example
........
IF obj.record * 1.20  < 500
.......
```

Example 2

The following is an example of using the **SELECT** statement in an expression within a condition. The trigger in this example uses the **SELECT** statement that contains an aggregate function **COUNT**(*) to compare the value with a constant. The **SELECT** statement must be enclosed in parentheses and must be placed at the end of the expression.

```
CREATE TRIGGER example
......
IF 1000 >  (SELECT COUNT( * ) FROM participant)
......
```

Caution

The expression given in the trigger condition may cause side effects on the database if a method is called while the condition is performed. A trigger condition must be constructed to avoid unexpected side effects in the database.

Correlation Name

You can access the column values defined in the target table by using a correlation name in the trigger definition. A correlation name is the instance that is actually affected by the database operation calling the trigger. A correlation name can also be specified in a trigger condition or action.

The types of correlation names are **new**, **old** and **obj**. These correlation names can be used only in instance triggers that have an **INSERT**, **UPDATE** or **DELETE** event.

As shown in the table below, the use of correlation names is further restricted by the event time defined for the trigger condition.

	BEFORE	**AFTER or DERERRED**
INSERT	**new**	**obj**
UPDATE	**obj** **new**	**obj** **old** (**AFTER**)
DELETE	**obj**	N/a

Correlation Name	**Representative Attribute Value**
obj	Refers to the current attribute value of an instance. This can be used to access attribute values before an instance is updated or deleted. It is also used to access attribute values after an instance has been updated or inserted.
new	Refers to the attribute value proposed by an insert or update operation. The new value can be accessed only before the instance is actually inserted or updated.
old	Refers to the attribute value that existed prior to the completion of an update operation. This value is maintained only while the trigger is being performed. Once the trigger is completed, the **old** values get lost.

TRIGGER Action

Description

A trigger action describes what to be performed if the trigger condition is true or omitted. If a specific point of time (**AFTER** or **DEFERRED**) is not given in the action clause, the action is executed at the same time as the trigger event.

The following is a list of actions that can be used for trigger definitions.

- **REJECT** : **REJECT** discards the operation that initiated the trigger and keeps the former state of the database, if the condition is not true. Once the operation is performed, **REJECT** is allowed only when the action time is **BEFORE** because the operation cannot be rejected. Therefore, you must not use **REJECT** if the action time is **AFTER** or **DERERRED**.
- **INVALIDATE TRANSACTION** : **INVALIDATE TRANSACTION** allows the event operation that called the trigger, but does not allow the transaction that contains the commit to be executed. You must cancel the transaction by using the **ROLLBACK** statement if it is not valid. Such action is used to protect the database from having invalid data after a data-changing event happens.
- **PRINT** : **PRINT** outputs trigger actions on the terminal screen in text messages, and can be used during developments or tests. The results of event operations are not rejected or discarded.
- **INSERT** : **INSERT** inserts one or more new instances to the table.
- **UPDATE** : **UPDATE** updates one or more column values in the table.
- **DELETE** : **DELETE** deletes one or more instances from the table.

Example

The following example shows how to define an action when a trigger is created. The medal_trig trigger defines **REJECT** in its action. **REJECT** can be specified only when the action time is **BEFORE**.

```
CREATE TRIGGER medal_trig
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
```

Caution

- Trigger may fall into an infinite loop when you use **INSERT** in an action of a trigger where an **INSERT** event is defined.
- If a trigger where an **UPDATE** event is defined runs on a partitioned table, you must be careful because the defined partition can be broken or unintended malfunction may occur. To prevent such situation, CUBRID outputs an error so that the **UPDATE** causing changes to the running partition is not executed. Trigger may fall into an infinite loop when you use **UPDATE** in an action of a trigger where an **UPDATE** event is defined.

ALTER TRIGGER

Description

In the trigger definition, **STATUS** and **PRIORITY** options can be changed by using the **ALTER** statement. If you need to alter other parts of the trigger (event targets or conditional expressions), you must delete and then re-create the trigger.

Syntax

```
ALTER TRIGGER trigger_name  trigger_option [ ; ]
trigger_option :
• STATUS { ACTIVE | INACTIVE }
• PRIORITY key
```

- *trigger_name* : Specifies the name of the trigger to be changed.
- *trigger_option* :
- **STATUS** { **ACTIVE** | **INACTIVE** } : Changes the status of the trigger.
- **PRIORITY** *key* : Changes the priority.

Example

The following is an example of creating the medal_trig trigger and then changing its state to **INACTIVE** and its priority to 0.7.

```
CREATE TRIGGER medal_trig
STATUS ACTIVE
BEFORE PDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
ALTER TRIGGER medal_trig STATUS INACTIVE;
ALTER TRIGGER medal_trig PRIORITY 0.7;
```

Caution

- Only one option can be specified in a single **ALTER TRIGGER** statement.
- To change a table trigger, you must be the trigger owner or granted the **ALTER** privilege on the table where the trigger belongs.
- A user trigger can only be changed by its owner. For more information on these options, see the CREATE TRIGGER (Syntax) section. The key specified together with the **PRIORITY** option must be a non-negative floating point value.

DROP TRIGGER

Description

You can drop a trigger by using the **DROP TRIGGER** statement.

Syntax

```
DROP TRIGGER trigger_name [ ; ]
```

- *trigger_name* : Specifies the name of the trigger to be dropped.

Example

The following is an example of dropping the medal_trig trigger.

```
DROP TRIGGER medal_trig;
```

Caution

- A user trigger (i.e. the trigger event is **COMMIT** or **ROLLBACK**) can be seen and dropped only by the owner.
- Only one trigger can be dropped by a single **DROP TRIGGER** statement. A table trigger can be dropped by a user who has an **ALTER** authorization on the table.

RENAME TRIGGER

Description

You can change a trigger name by using the **TRIGGER** reserved word in the **RENAME** statement.

Syntax

```
RENAME TRIGGER old_trigger_name AS new_trigger_name [ ; ]
```

- *old_trigger_name* : Specifies the current name of the trigger.
- *new_trigger_name* : Specifies the name of the trigger to be changed.

Example

```
RENAME TRIGGER medal_trigger AS medal_trig;
```

Caution

- A trigger name must be unique among all trigger names. The name of a trigger can be the same as the table name in the database.
- To rename a table trigger, you must be the trigger owner or granted the **ALTER** privilege on the table where the trigger belongs. A user trigger can only be renamed by its user.

Deferred Condition and Action

Definition

A deferred trigger action and condition can be executed later or canceled. These triggers include a **DEFERRED** time option in the event time or action clause. If the **DEFERRED** option is specified in the event time and the time is omitted before the action, the action is deferred automatically.

Executing Deferred Condition and Action

Description

Executes the deferred condition or action of a trigger immediately.

Syntax

```
EXECUTE DEFERRED TRIGGER trigger_identifier [ ; ]
trigger_identifier :
• trigger_name
• ALL TRIGGERS
```

- *trigger_identifier* :
- *trigger_name* : Executes the deferred action of the trigger when a trigger name is specified.
- **ALL TRIGGERS** : All currently deferred actions are executed.

Dropping Deferred Condition and Action

Description

Drops the deferred condition and action of a trigger.

Syntax

```
DROP DEFERRED TRIGGER trigger_identifier [ ; ]
trigger_option :
• trigger_name
• ALL TRIGGERS
```

- *trigger_option* :
- *trigger_name* : Cancels the deferred action of the trigger when a trigger name is specified.
- **ALL TRIGGERS** : All currently deferred actions are canceled.

Granting TRIGGER Authorization

Description

Trigger authorization is not granted explicitly. Authorization on the table trigger is automatically granted to the user if the authorization is granted on the event target table described in the trigger definition. In other words, triggers that have table targets (**INSERT**, **UPDATE**, etc.) are seen by all users. User triggers (**COMMIT** and **ROLLBACK**) are seen only by the user who defined the triggers. All authorizations are automatically granted to the trigger owner.

Caution

- To define a table trigger, you must have an **ALTER** authorization on the table.

- To define a user trigger, the database must be accessed by a valid user.

Trigger on REPLACE and INSERT … ON DUPLICATE KEY UPDATE

Deferred Actions

Description

When the **REPLACE** statement and **INSERT … ON DUPLICATE KEY UPDATE** statement are executed, the trigger is executed in CUBRID, while **DELETE**, **UPDATE**, **INSERT** jobs occur internally. The following table shows the order in which the trigger is executed in CUBRID depending on the event that occurred when the **REPLACE** or **INSERT … ON DUPLICATE KEY UPDATE** statement is executed. Both the **REPLACE** statement and **INSERT … ON DUPLICATE KEY UPDATE** statement do not execute triggers in the inherited class (table).

Execution Sequence of Triggers in the REPLACE and the INSERT … ON DUPLICATE KEY UPDATE statements

Event	Execution Sequence of Triggers
REPLACE When a record is deleted and new one is inserted	BEFORE DELETE > AFTER DELETE > BEFORE INSERT > AFTER INSERT
INSERT … ON DUPLICATE KEY UPDATE When a record is updated	BEFORE UPDATE > AFTER UPDATE
REPLACE, INSERT … ON DUPLCATE KEY UPDATE Only when a record is inserted	BEFORE INSERT > AFTER INSERT

Example

The folllowing is an example in which the trigger inserts records to the trigger table if **INSERT … ON DUPLICATE KEY UPDATE** and **RELPACE** are executed in the with_trigger table.

```
CREATE TABLE with_trigger (id INT UNIQUE);
INSERT INTO with_trigger VALUES (11);

CREATE TABLE trigger_actions (val INT);

CREATE TRIGGER trig_1 BEFORE INSERT ON with_trigger EXECUTE INSERT INTO trigger_actions
VALUES (1);
CREATE TRIGGER trig_2 BEFORE UPDATE ON with_trigger EXECUTE INSERT INTO trigger_actions
VALUES (2);
CREATE TRIGGER trig_3 BEFORE DELETE ON with_trigger EXECUTE INSERT INTO trigger_actions
VALUES (3);

INSERT INTO with_trigger VALUES (11) ON DUPLICATE KEY UPDATE id=22;

SELECT * FROM trigger_actions;
          va
==============
            2

REPLACE INTO with_trigger VALUES (22);

SELECT * FROM trigger_actions;
          va
==============
            2
            3
            1
```

TRIGGER Debugging

Definition and Example

Description

Once a trigger is defined, it is recommended to check whether it is running as intended. Sometimes the trigger takes more time than expected in processing. This means that it is adding too much overhead to the system or has fallen into a recursive loop. This section explains several ways to debug the trigger.

Example

The following is an example of a trigger that was defined to fall into a recursive loop when it is called. A loop trigger is somewhat artificial in its purpose, but can be used as an example for debugging the trigger.

```
CREATE TRIGGER loop_tgr
BEFORE UPDATE ON participant(gold)
IF new.gold > 0
EXECUTE UPDATE participant
        SET gold = new.gold - 1
        WHERE nation_code = obj.nation_code AND host_year = obj.host_year;
```

Viewing TRIGGER Execution Log

Description

You can view the execution log of the trigger from a terminal by using the **SET TRIGGER TRACE** statement.

Syntax

```
SET TRIGGER TRACE switch [ ; ]
switch:
• ON
• OFF
```

- *switch* :
- **ON** : Runs the **TRACE** until the switch is set to **OFF** or the current database session terminates.
- **OFF** : Stops the **TRACE**.

Example

The following is an example of running the **TRACE** and executing the loop trigger to view the trigger execution logs. To identify the trace for each condition and action executed when the trigger is called, a message is displayed on the terminal. The following message appears 15 times because the loop trigger is executed until the gold value becomes 0.

```
SET TRIGGER TRACE ON;

UPDATE participant SET gold =15 WHERE nation_code = 'KOR' AND host_year = 1988;
TRACE: Evaluating condition for trigger "loop".
TRACE: Executing action for trigger "loop".
```

Limiting Nested TRIGGER

Description

With the **MAXIMUM DEPTH** keyword of the **SET TRIGGER** statement, you can limit the number of triggers to be initiated at each step. By doing so, you can prevent a recursively called trigger from falling into an infinite loop.

Syntax

```
SET TRIGGER [ MAXIMUM ] DEPTH count [ ; ]
count:
• unsigned_integer_literal
```

- *unsigned_integer_literal* : A positive integer value that specifies the number of times that a trigger can recursively start another trigger or itself. If the number of triggers reaches the maximum depth, the database request stops(aborts) and the transaction is marked as invalid. The specified **DEPTH** applies to all other triggers except for the current session. The maximum value is 32.

Example

The following is an example of setting the maximum number of times of recursive trigger calling to 10. This applies to all triggers that start subsequently. In this example, the gold column value is updated to 15, so the trigger is called 16 times in total. This exceeds the currently set maximum depth and the following error message occurs.

```
SET TRIGGER MAXIMUM DEPTH 10;
UPDATE participant SET gold = 15 WHERE nation_code = 'KOR' AND host_year = 1988;

ERROR: Maximum trigger depth 10 exceeded at trigger "loop_tgr".
```

TRIGGER Example

Description

This section covers trigger definitions in the demo database. The triggers created in the demodb database are not complex, but use most of the features available in CUBRID. If you want to maintain the original state of the demodb database when testing such triggers, you must perform a rollback after changes are made to the data.

Triggers created by the user in the own database can be as powerful as applications created by the user.

Example 1

The following trigger created in the participant table rejects an update to the medal column (gold, silver, bronze) if a given value is smaller than 0. The evaluation time must be **BEFORE** because a correlation name new is used in the trigger condition. Although not described, the action time of this trigger is also **BEFORE**.

```
CREATE TRIGGER medal_trigger
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
```

The medal_trigger trigger starts when the number of gold medals of the country whose nation code is 'BLA' is updated. Since a negative value is not permitted for the number of gold medals as shown above, this update is not allowed.

```
UPDATE participant
SET gold = -10
WHERE nation_code = 'BLA';
```

Example 2

The following trigger has the same condition as the one above except that **STATUS INACTIVE** is added. If the STATUS statement is omitted, the default value is **ACTIVE**. You can change the status to **INACTIVE** by using the **ALTER TRIGGER** statement.

You can specify whether or not to execute the trigger depending on the **STATUS** value.

```
CREATE TRIGGER medal_trig
STATUS ACTIVE
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
ALTER TRIGGER medal_trig
STATUS INACTIVE;
```

Example 3

The following trigger shows how integrity constraint is enforced when a transaction is committed. This example is different from the previous ones, in that one trigger can have specific conditions for multiple tables.

```
CREATE TRIGGER check_null_first
```

```
BEFORE COMMIT
IF 0 < (SELECT count(*) FROM athlete WHERE gender IS NULL)
OR 0 < (SELECT count(*) FROM game WHERE nation_code IS NULL)
EXECUTE REJECT;
```

Example 4

The following trigger delays the update integrity constraint check for the record table until the transaction is committed. Since the **DEFERRED** keyword is given as the event time, the trigger is not executed at the time.

```
CREATE TRIGGER deferred_check_on_record
DEFERRED UPDATE ON record
IF obj.score = '100'
EXECUTE INVALIDATE TRANSACTION;
```

Once completed, the update in the record table can be confirmed at the last point (commit or rollback) of the current transaction. The correlation name old cannot be used in the conditional clause of the trigger where **DEFERRED UPDATE** is used. Therefore, you cannot create a trigger as the following.

```
CREATE CLASS foo (n int);
CREATE TRIGGER foo_trigger
    DEFERRED UPDATE ON foo
    IF old.n = 100
    EXECUTE PRINT 'foo_trigger';
```

If you try to create a trigger as shown above, an error message is displayed and the trigger fails.

```
ERROR: Error compiling condition for 'foo_trigger' : old.n is not defined
```

The correlation name **old** can be used only with **AFTER**.

Java Stored Function/Procedure

Overview

Stored functions and procedures are used to implement complicated program logic that is not possible with SQL. They allow users to manipulate data more easily. Stored functions/procedures are blocks of code that have a flow of commands for data manipulation and are easy to manipulate and administer.

CUBRID supports to develop stored functions and procedures in Java. Java stored functions/procedures are executed on the JVM (Java Virtual Machine) hosted by CUBRID.

You can call Java stored functions/procedures from SQL statements or from Java applications using JDBC.

The advantages of using Java stored functions/procedures are as follows:

- **Productivity and usability** : Java stored functions/procedures, once created, can be reused anytime. They can be called from SQL statements or from Java applications using JDBC.
- **Excellent interoperability and portability** : Java stored functions/procedures use the Java Virtual Machine. Therefore, they can be used on any system where the Java Virtual Machine is available.

Environment Configuration for Java Stored Function/Procedure

To use Java-stored functions/procedures in CUBRID, you must have JRE (Java Runtime Environment) 1.6 or better installed in the environment where the CUBRID server is installed. You can download JRE from the Developer Resources for Java Technology (http://java.sun.com).

If the java_stored_procedure parameter in the CUBRID configuration file (cubrid.conf) is set to yes, CUBRID 64-bit needs a 64-bit Java Runtime Environment, and CUBRID 32-bit needs a 32-bit Java Runtime Environment. For example, when you run CUBRID 64-bit in the system in which a 32-bit JAVA Runtime Environment is installed, the following error may occur.

```
% cubrid server start demodb

This may take a long time depending on the amount of recovery works to do.
WARNING: Java VM library is not found :
/usr/java/jdk1.6.0_15/jre/lib/amd64/server/libjvm.so: cannot open shared object file: No
such file or directory.
Consequently, calling java stored procedure is not allowed
```

Execute the following command to check the JRE version if you have it already installed in the system.

```
% java -version Java(TM) SE Runtime Environment (build 1.6.0_05-b13)
Java HotSpot(TM) 64-Bit Server VM (build 10.0-b19, mixed mode)
```

Windows Environment

For Windows, CUBRID loads the **jvm.dll** file to run the Java Virtual Machine. CUBRID first locates the **jvm.dll** file from the **PATH** environment variable and then loads it. If it cannot find the file, it uses the Java runtime information registered in the system registry.

You can configure the **JAVA_HOME** environment variable and add the directory in which the Java executable file is located to **Path**, by executing the command as follows: For information on configuring environment variables using GUI, see Setting up the JDBC Environment.

- An example of installing 64 Bit JDK 1.6 and configuring the environment variables

```
% set JAVA_HOME=C:\jdk1.6.0
% set PATH=%PATH%;%JAVA_HOME%\jre\bin\server
```

- An example of installing 32 Bit JDK 1.6 and configuring the environment variables

```
% set JAVA_HOME=C:\jdk1.6.0
% set PATH=%PATH%;%JAVA_HOME%\jre\bin\client
```

To use other vendor's implementation instead of Sun's Java Virtual Machine, add the path of the **jvm.dll** file to the **PATH** variable during the installation.

Linux/UNIX Environment

For Linux/UNIX environment, CUBRID loads the **libjvm.so** file to run the Java Virtual Machine. CUBRID first locates the **libjvm.so** file from the **LD_LIBRARY_PATH** environment variable and then loads it. If it cannot find the file, it uses the **JAVA_HOME** environment variable. For Linux, glibc version 2.3.4 or higher is supported. The following is an example of configuring the Linux environment variable (e.g., **.profile**, **.cshrc**, **.bashrc**, **.bash_profile**, etc.).

- An example of installing 64 Bit JDK 1.6 and configuring the environment variables in a bash shell

```
% JAVA_HOME=/usr/java/jdk1.6.0_10
%
LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/amd64:$JAVA_HOME/jre/lib/amd64/server:$LD_LIBRARY_PA
TH
% export JAVA_HOME
% export LD_LIBRARY_PATH
```

- An example of installing 32 Bit JDK 1.6 and configuring the environment variables in a bash shell

```
% JAVA_HOME=/usr/java/jdk1.6.0_10
%
LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/i386/:$JAVA_HOME/jre/lib/i386/client:$LD_LIBRARY_PAT
H
% export JAVA_HOME
% export LD_LIBRARY_PATH
```

- An example of installing 64 Bit JDK 1.6 and configuring the environment variables in a csh

```
% setenv JAVA_HOME /usr/java/jdk1.6.0_10
% setenv LD_LIBRARY_PATH
$JAVA_HOME/jre/lib/amd64:$JAVA_HOME/jre/lib/amd64/server:$LD_LIBRARY_PATH
% set path=($path $JAVA_HOME/bin .)
```

- An example of installing 32 Bit JDK 1.6 and configuring the environment variables in a csh shell

```
% setenv JAVA_HOME /usr/java/jdk1.6.0_10
% setenv LD_LIBRARY_PATH
$JAVA_HOME/jre/lib/i386:$JAVA_HOME/jre/lib/i386/client:$LD_LIBRARY_PATH
% set path=($path $JAVA_HOME/bin .)
```

To use other vendor's implementation instead of Sun's Java Virtual Machine, add the path of the JVM (**libjvm.so**) to the library path during the installation.

The path of the **libjvm.so** file can be different depending on the platform. For example, the path is the **$JAVA_HOME/jre/lib/sparc** directory in a SUN Sparc machine.

How to Write Java Stored Function/Procedure

Steps to write a Java stored function/procedure are as follows:

- Check the cubrid.conf file
- Write and compile the Java source code
- Load the complied Java class into CUBRID
- Publish the loaded Java class
- Call the Java stored function/procedure

Check the cubrid.conf file

By default, the **java_stored_procedure** is set to **no** in the **cubrid.conf** file. To use a Java stored function/procedure, this value must be changed to **yes**. For more information on this value, see Other Parameters in Database Server Configuration.

Write and compile the Java source code

Compile the SpCubrid.java file as follows:

```
public class SpCubrid{
      public static String HelloCubrid() {
              return "Hello, Cubrid !!";
      }
      public static int SpInt(int i) {
              return i + 1;
      }
      public static void outTest(String[] o) {
              o[0] = "Hello, CUBRID";
      }
}

%javac SpCubrid.java
```

Here, the Java class method must be public static.

Load the compiled Java class into CUBRID

Load the compiled Java class into CUBRID.

```
% loadjava demodb SpCubrid.class
```

Publish the loaded Java class

Create a CUBRID stored function and publish the Java class as shown below.

```
csql> create function hello() return string
csql> as language java
csql> name 'SpCubrid.HelloCubrid() return java.lang.String';
```

Call the Java stored function/procedure

Call the published Java stored function as follows:

```
csql> call hello() into :Hello;

 Result
======================
'Hello, Cubrid !!'
```

Using Server-side Internal JDBC Driver

To access the database from a Java stored function/procedure, you must use the server-side JDBC driver. As Java stored functions/procedures are executed within the database, there is no need to make the connection to the server-side JDBC driver again. To acquire a connection to the database using the server-side JDBC driver, you can either use "**jdbc:default:connection:**" as the URL for JDBC connection, or call the **getDefaultConnection**() method of the **cubrid.jdbc.driver.CUBRIDDriver** class.

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
Connection conn =    DriverManager.getConnection("jdbc:default:connection:");
```

or

```
cubrid.jdbc.driver.CUBRIDDriver.getDefaultConnection();
```

If you connect to the database using the JDBC driver as shown above, the transaction in the Java stored function/procedure is ignored. That is, database operations executed in the Java stored function/procedure belong to the transaction that called the Java stored function/procedure. In the following example, **conn.commit()** method of the **Athlete** class is ignored.

```
import java.sql.*;
public class Athlete{
    public static void Athlete(String name, String gender, String nation_code, String
event) throws SQLException{
        String sql="INSERT INTO ATHLETE(NAME, GENDER, NATION_CODE, EVENT)" + "VALUES
(?, ?, ?, ?)";
        try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            Connection conn = DriverManager.getConnection("jdbc:default:connection:");
            PreparedStatement pstmt = conn.prepareStatement(sql);
```

```
            pstmt.setString(1, name);
            pstmt.setString(2, gender);
            pstmt.setString(3, nation_code);
            pstmt.setString(4, event);;
            pstmt.executeUpdate();

            pstmt.close();
            conn.commit();
            conn.close();
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Connecting to Other Database

You can connect to another outside database instead of the currently connected one even when the server-side JDBC driver is being used. Acquiring a connection to an outside database is not different from a generic JDBC connection. For more information, see JDBC API.

If you connect to other databases, the connection to the CUBRID database does not terminate automatically even when the execution of the Java method ends. Therefore, the connection must be explicitly closed so that the result of transaction operations such as **COMMIT** or **ROLLBACK** will be reflected in the database. That is, a separate transaction will be performed because the database that called the Java stored function/procedure is different from the one where the actual connection is made.

```
import java.sql.*;
public class SelectData {
  public static void SearchSubway(String[] args) throws Exception {
Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;
  try {
    Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
    conn =
DriverManager.getConnection("jdbc:CUBRID:localhost:33000:demodb:::","","");
    String sql = "select line_id, line from line";
    stmt = conn.createStatement();
    rs = stmt.executeQuery(sql);
    while(rs.next()) {
      int host_year = rs.getString("host_year");
      String host_nation = rs.getString("host_nation");
      System.out.println("Host Year ==> " + host_year);
      System.out.println(" Host Nation==> " + host_nation);
      System.out.println("\n=========\n");
    }
    rs.close();
    stmt.close();
    conn.close();
    } catch ( SQLException e ) {
         System.err.println(e.getMessage());
    } catch ( Exception e ) {
         System.err.println(e.getMessage());
    } finally {
          if ( conn != null ) conn.close();
    }
  }
}
```

When the Java stored function/procedure being executed should run only on JVM located in the database server, you can check where it is running by calling System.getProperty ("cubrid.server.version") from the Java program source. The result value is the database version if it is called from the database; otherwise, it is **NULL**.

loadjava Utility

Description

To load a compiled Java or JAR (Java Archive) file into CUBRID, use the **loadjava** utility. If you load a Java *.class or *.jar file using the **loadjava** utility, the file is moved to the specified database path.

Syntax

```
loadjava <option> database-name java-class-file
```

- *database-name* : The name of the database where the Java file is to be loaded.
- *java-class-file* : The name of the Java class or jar file to be loaded.
- <*option*> :
- **-y** : Automatically overwrites a class file with the same name, if any. The default value is **no**. If you load the file without specifying the **-y** option, you will be prompted to ask if you want to overwrite the class file with the same name (if any).

Loaded Java Class Publish

Overview

In CUBRID, it is required to publish Java classes to call Java methods from SQL statements or Java applications. You must publish Java classes by using call specifications because it is not known how a function in a class will be called by SQL statements or Java applications when Java classes are loaded.

Call Specifications

To use a Java stored function/procedure in CUBRID, you must write call specifications. With call specifications, Java function names, parameter types, return values and their types can be accessed by SQL statements or Java applications. To write call specifications, use **CREATE FUNCTION** or **CREATE PROCEDURE** statement. Java stored function/procedure names are not case sensitive. The maximum number of characters a Java stored function/procedure can have is 256. The maximum number of parameters a Java stored function/procedure can have is 64.

Syntax

```
CREATE {PROCEDURE procedure_name[(param[, param]...] | FUNCTION function_name[(param[,
param]...] RETURN sql_type }
{IS | AS} LANGUAGE JAVA
NAME 'method_fullname (java_type_fullname[,java_type_fullname]... [return
java_type_fullname]';

parameter_name [IN|OUT|IN OUT|INOUT] sql_type
   (default IN)
```

If the parameter of a Java stored function/procedure is set to **OUT**, it will be passed as a one-dimensional array whose length is 1. Therefore, a Java method must store its value to pass in the first space of the array.

Example

```
CREATE FUNCTION Hello() RETURN VARCHAR
AS LANGUAGE JAVA
NAME 'SpCubrid.HelloCubrid() return java.lang.String';
CREATE FUNCTION Sp_int(i int) RETURN int
AS LANGUAGE JAVA
NAME 'SpCubrid.SpInt(int) return int';

CREATE PROCEDURE Phone_Info(name varchar, phoneno varchar)
AS LANGUAGE JAVA
NAME 'PhoneNumber.Phone(java.lang.String, java.lang.String)';
```

When a Java stored function/procedure is published, it is not checked whether the return definition of the Java stored function/procedure coincides with the one in the declaration of the Java file. Therefore, the Java stored function/procedure follows the *sql_type* return definition provided at the time of registration. The return definition in the declaration is significant only as user-defined information.

Data Type Mapping

In call specifications, the data types SQL must correspond to the data types of Java parameter and return value. The following table shows SQL/Java data types allowed in CUBRID.

Data Type Mapping

SQL Type	Java Type
CHAR, VARCHAR	java.lang.String, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.math.BigDecimal, byte, short, int, long, float, double
NUMERIC, SHORT, INT, FLOAT, DOUBEL, CURRENCY	java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.math.BigDecimal, java.lang.String, byte, short, int, long, float, double
DATE, TIME, TIMESTAMP	java.sql.Date, java.sql.Time, java.sql.Timestamp, java.lang.String
SET, MULTISET, SEQUENCE	java.lang.Object[], java primitive type array, java.lang.Integer[] ...
OBJECT	cubrid.sql.CUBRIDOID
CURSOR	cubrid.jdbc.driver.CUBRIDResultSet

Checking the Published Java Stored Function/Procedure Information

You can check the information on the published Java stored function/procedure The **db_stored_procedure** system virtual table provides virtual table and the **db_stored_procedure_args** system virtual table. The **db_stored_procedure** system virtual table provides the information on stored names and types, return types, number of parameters, Java class specifications, and the owner. The **db_stored_procedure_args** system virtual table provides the information on parameters used in the stored function/procedure.

```
SELECT * from db_stored_procedure;
sp_name     sp_type   return_type    arg_count
sp_name               sp_type               return_type            arg_count  lang
target                owner
================================================================================
'hello'               'FUNCTION'            'STRING'                       0  'JAVA''SpCu
brid.HelloCubrid() return java.lang.String'  'DBA'

'sp_int'              'FUNCTION'            'INTEGER'                      1  'JAVA''SpCu
brid.SpInt(int) return int'  'DBA'

'athlete_add'         'PROCEDURE'           'void'                         4  'JAVA''Athl
ete.Athlete(java.lang.String, java.lang.String, java.lang.String,
java.lang.String)'  'DBA'
SELECT * from db_stored_procedure_args;
sp_name   index_of  arg_name  data_type      mode
=================================================
 'sp_int'                      0  'i'                   'INTEGER'             'IN'
 'athlete_add'                 0  'name'                'STRING'              'IN'
 'athlete_add'                 1  'gender'              'STRING'              'IN'
 'athlete_add'                 2  'nation_code'         'STRING'              'IN'
 'athlete_add'                 3  'event'               'STRING'              'IN'
```

Deleting Java Stored Functions/Procedures

You can delete published Java stored functions/procedures in CUBRID. To delete a Java function/procedure, use the **DROP FUNCTION** *function_name* or **DROP PROCEDURE** *procedure_name* statement. Also, you can delete multiple Java stored functions/procedures at a time with several *function_name*s or *procedure_name*s separated by a comma (,).

A Java stored function/procedure can be deleted only by the user who published it or by DBA members. For example, if a **PUBLIC** user published the 'sp_int' Java stored function, only the **PUBLIC** or **DBA** members can delete it.

```
drop function hello[, sp_int]
drop procedure Athlete_Add
```

Java Stored Function/Procedure Call

Using CALL Statement

You can call the Java stored functions/procedures by using a **CALL** statement, from SQL statements or Java applications.

The following shows how to call them by using the **CALL** statement. The name of the Java stored function/procedure called from a **CALL** statement is not case sensitive.

Syntax

```
CALL {procedure_name ([param[, param]...) | function_name ([param[, param]...)
INTO :host_variable
param {literal | :host_variable}
```

Example

```
call Hello() into :HELLO;
call Sp_int(3) into :i;
call phone_info('Tom','016-111-1111');
```

In CUBRID, the Java functions/procedures are called by using the same **CALL** statement. Therefore, the **CALL** statement is processed as follows:

- It is processed as a method if there is a target class in the **CALL** statement.
- If there is no target class in the **CALL** statement, it is checked whether a Java stored function/procedure is executed or not; a Java stored function/procedure will be executed if one exists.
- If no Java stored function/procedure exists in step 2 above, it is checked whether a method is executed or not; a method will be executed if one with the same name exists.

The following error occurs if you call a Java stored function/procedure that does not exist.

```
CALL deposit()
ERROR: Stored procedure/function 'deposit' is not exist.

CALL deposit('Tom', 3000000)
ERROR: Methods require an object as their target.
```

If there is no argument in the **CALL** statement, a message "ERROR: Stored procedure/function 'deposit' is not exist." appears because it can be distinguished from a method. However, if there is an argument in the **CALL** statement, a message "ERROR: Methods require an object as their target." appears because it cannot be distinguished from a method.

If the **CALL** statement is nested within another **CALL** statement calling a Java stored function/procedure, or if a subquery is used in calling the Java function/procedure, the **CALL** statement is not executed.

```
call phone_info('Tom', call sp_int(999));
call phone_info((select * from Phone where id='Tom'));
```

If an exception occurs during the execution of a Java stored function/procedure, the exception is logged and stored in the *dbname*_**java.log** file. To display the exception on the screen, change a handler value of the **$CUBRID/java/logging.properties** file to " java.lang.logging.ConsoleHandler." Then, the exception details are displayed on the screen.

Calling from SQL Statement

You can call a Java stored function from a SQL statement as shown below.

```
select Hello() from db_root;
select sp_int(99) from db_root;
```

You can use a host variable for the IN/OUT data type when you call a Java stored function/procedure as follows:

```
SELECT 'Hi' INTO :out_data FROM db_root;
CALL test_out(:out_data);
SELECT :out_data FROM db_root;
```

The first clause calls a Java stored procedure in out mode by using a parameter variable; the second is a query clause retrieving the assigned host variable out_data.

Calling from Java Application

To call a Java stored function/procedure from a Java application, use a **CallableStatement** object.

Create a phone class in the CUBRID database.

```
CREATE TABLE phone(
     name varchar(20),
     phoneno varchar(20)
)
```

Compile the following PhoneNumber.java file, load the Java class file into CUBRID, and publish it.

```
import java.sql.*;
import java.io.*;
public class PhoneNumber{    public static void Phone(String name, String phoneno) throws
Exception{
        String sql="INSERT INTO PHONE(NAME, PHONENO)"+ "VALUES (?, ?)";
        try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            Connection conn = DriverManager.getConnection("jdbc:default:connection:");
            PreparedStatement pstmt = conn.prepareStatement(sql);

            pstmt.setString(1, name);
            pstmt.setString(2, phoneno);
            pstmt.executeUpdate();
            pstmt.close();
            conn.commit();
            conn.close();
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
create PROCEDURE phone_info(name varchar, phoneno varchar)
as language java
name 'PhoneNumber.Phone(java.lang.String, java.lang.String)';
```

Create and run the following Java application.

```
import java.sql.*;
public class StoredJDBC{
    public static void main(){
        Connection conn = null;
        Statement stmt= null;
        int result;
        int i;
        try{
 Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            conn =
DriverManager.getConnection("jdbc:CUBRID:localhost:33000:subway:::","","");
            CallableStatement cs;
            cs = conn.prepareCall("call PHONE_INFO(?, ?)");
            cs.setString(1, "Jane");
            cs.setString(2, "010-1111-1111");
            cs.executeUpdate();
            conn.commit();
            cs.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }}
```

Retrieve the phone class after executing the program above; the following result would be displayed.

```
SELECT * from phone;
name                  phoneno
============================================
    'Jane'                '010-111-1111'
```

Note

Return Value of Java Stored Function/Procedure and Precision Type on IN/OUT

To limit the return value of Java stored function/procedure and precision type on IN/OUT, CUBRID processes as follows:

Checks the sql_type of the Java stored function/procedure.

Passes the value returned by Java to the database with only the type converted if necessary, ignoring the number of digits defined during creating the Java stored function/procedure. In principle, the user manipulates the passed data directly in the database.

Take a look at the following **typestring**() Java stored function.

```
public class JavaSP1{
    public static String typestring(){
        String temp = " ";
        for(int i=0 i< 1 i++)
            temp = temp + "1234567890";
        return temp;
}
}

create function typestring() return char(5)
as language java
name 'JavaSP1.typestring() return java.lang.String';

Call typestring()

  Result
======================
  ' 1234567890'
```

Returning java.sql.ResultSet in Java Stored Procedure

In CUBRID, you must use **CURSOR** as the data type when you declare a Java stored function/procedure that returns a **java.sql.ResultSet**.

```
create function rset() return cursor
as language java
name 'JavaSP2.TResultSet() return java.sql.ResultSet'
```

Before the Java file returns **java.sql.ResultSet**, it is required to cast to the **CUBRIDResultSet** class and then to call the **setReturnable**() method.

```
public static class JavaSP2 {
public static ResultSet TResultSet(){
    try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    ((CUBRIDConnection)con).setCharset("euc_kr");
    String sql = "select * from station";
    Statement stmt=con.createStatement();
    ResultSet rs = stmt.executeQuery(sql);
    ((CUBRIDResultSet)rs).setReturnable();
    return rs;
    } catch (Exception e) {
              e.printStackTrace();
    }
    return null;
  }
```

```
}
```

In the calling block, you must set the OUT argument with **Types.JAVA_OBJECT**, get the argument to the **getObject**() function, and then cast it to the **java.sql.ResultSet** type before you use it. In addition, the **java.sql.ResultSet** is only available to use in **CallableStatement** of JDBC.

```
import java.sql.*;
public class TestResultSet{
  public static void main(String[] args) {
    Connnection conn = null;
    Statement stmt= null;
    int result;
    int i;

    try{
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
    conn = DriverManager.getConnection("jdbc:CUBRID:localhost:33000:demodb:::","","");

    CallableStatement cstmt = con.prepareCall("?=CALL rset()");
    cstmt.registerOutParameter(1, Types.JAVA_OBJECT);
    cstmt.execute();
    ResultSet rs = (ResultSet) cstmt.getObject(1);
    while(rs.next()) {
      System.out.println(rs.getString(1));
    }
      rs.close();
     } catch (Exception e) {
             e.printStackTrace();
    }
}
```

You cannot use the **ResultSet** as an input argument. If you pass it to an IN argument, an error occurs. An error also occurs when calling a function that returns **ResultSet** in a non-Java environment.

IN/OUT of Set Type in Java Stored Function/Procedure

If the set type of the Java stored function/procedure in CUBRID is IN OUT, the value of the argument changed in Java must be applied to IN OUT. When the set type is passed to the OUT argument, it must be passed as a two-dimensional array.

```
Create procedure setoid(x in out set, z object)
as language java name
'SetOIDTest.SetOID(cubrid.sql.CUBRIDOID[][], cubrid.sql.CUBRIDOID';

public static void SetOID(cubrid.sql.CUBRID[][] set, cubrid.sql.CUBRIDOID aoid){
  Connection conn=null;
  Statement stmt=null;
  String ret="";
  Vector v = new Vector();
  cubrid.sql.CUBRIDOID[] set1 = set[0];
  try {
    if(set1!=null) {
      int len = set1.length;
      int i = 0;
      for (i=0 i< len i++)
        v.add(set1[i]);
    }
  v.add(aoid);
  set[0]=(cubrid.sql.CUBRIDOID[]) v.toArray(new cubrid.sql.CUBRIDOID[]{});
  } catch(Exception e) {
    e.printStackTrace();
    System.err.pirntln("SQLException:"+e.getMessage());
  }
}
```

Using OID in Java Stored Function/Procedure

In case of using the OID type value for IN/OUT in CUBRID, use the value passed from the server.

```
create procedure tOID(i inout object, q string)
as language java
```

```
name 'OIDtest.tOID(cubrid.sql.CUBRIDOID[], java.lang.String)';

public static void tOID(CUBRIDOID[] oid, String query)
{
  Connection conn=null;
  Statement stmt=null;
  String ret="";

  try {
    Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
    conn=DriverManager.getConnection("jdbc:default:connection:");

    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    System.out.println("query:"+ query);
    while(rs.next()) {
      oid[0]=(CUBRIDOID)rs.getObject(1);
      System.out.println("oid:"+oid[0].getTableName());
    }
    stmt.close();
    conn.close();
  } catch (SQLException e) {
    e.printStackTrace();
    System.err.println("SQLException:"+e.getMessage());
  } catch (Exception e) {
    e.printStackTrace();
    system.err.println("Exception:"+ e.getMessage());
  }
}
```

METHOD

Overview

This chapter describes methods (software routines) that extend or customize the features of the CUBRID database system.

The methods are written in C and called by the **CALL** or **EVALUATE** statement. A method program is loaded and linked with the application currently running by the dynamic loader when the method is called. The return value created as a result of the method execution is passed to the caller.

This chapter describes the following topics:

- Method Types
- Calling a Method

METHOD Type

The CSQL language supports the following two types of methods: class and instance methods.

- The **class method** is a method called by a class object. It is usually used to create a new class instance or to initialize it. It is also used to access or update class attributes.
- The **instance method** is a method called by a class instance. It is used more often than the class method because most operations are executed in the instance. For example, an instance method can be written to calculate or update the instance attribute. This method can be called from any instance of the class in which the method is defined or of the subclass that inherits the method.

The method inheritance rules are similar to those of the attribute inheritance. The subclass inherits classes and instance methods from the super class. The subclass has only the name of a class or instance method definition inherited from the super class.

The rules for resolving method name conflicts are same as those for attribute name conflicts. For more information about attribute/method inheritance conflicts, see Overview in Class Conflict Resolution.

Calling METHOD

Overview

Methods are executed by the **CALL** or **EVALUATE** statement, and their results are returned the same way as the query results.

These statements are also used to call a method from a query. (The **CALL** or **EVALUATE** keyword is omitted.)

CALL Statement

Description

In CUBRID, the **CALL** statement is used to call a method defined in the database. Both table and record methods can be called by the **CALL** statement.

Syntax

```
CALL method_call [ ; ]
method_call :
• method_name ( [ arg_value [ {, arg_value }_ ] ] ) ON call_target [ to_variable ]
• method_name ( call_target [, arg_value [ {, arg_value }_ ] ] ) [ to_variable ]
arg_value :
• any CSQL expression
call_target :
• an object-valued expression
```

```
to_variable :
• INTO variable
• TO variable
```

- The *method_name* is either the method name defined in the table or the system-defined method name provided with CUBRID. A method requires one or more parameters. If there is no parameter for the method, a set of blank parentheses must be used.
- *call_target* can use an object-valued expression that contains a class name, a variable, another method call (which returns an object). To call a class method for a class object, you must place the **CLASS** keyword before the *call_target*. In this case, the table name must be the name of the class where the table method is defined. To call a record method, you must specify the expression representing the record object. You can optionally store the value returned by the table or record method in the *to_variable*. This returned variable value can be used in the **CALL** statement just like the *call_target* or *arg_value* parameter.
- Calling nested methods is possible when other *method_call* is the *call_target* of the method or given as one of the *arg_value* parameters.

EVALUATE Statement

Description

The **EVALUATE** statement is also used to call a method defined in the database.

In the **EVALUATE** statement, a method call is a *term* in an expression. If the method returns a constant value, another constant (or a method returning a constant) can also be a term in an expression. Both class and instance methods can be called by the **EVALUATE** statement.

Syntax

```
EVALUATE expression [ ; ]

expression:
• [ + | - ] term [ { + | - | * | / } term ]

term:
• method_call

method_call :
• method_name ( call_target [, arg_value [ {, arg_value }_ ] ] ) [ to_variable ]
            method_name ( [ arg_value [ {, arg_value }_ ] ] )
            ON call_target [ to_variable ]
arg_value :
• literal
• variable
• expression

call_target :
• CLASS class_name
• variable
• expression
• method_call

to_variable :
• INTO variable
• TO variable
```

In the **EVALUATE** statement, the target argument for the specified method is represented in the parentheses following the *method_name*. The target can be the first field in the list, followed by method arguments. If the method executed is a class method, the **CLASS** keyword must precede the target class as the first field in the list. If only the method arguments are included in the parentheses, the *call_target* should be in the **ON** clause.

The **EVALUATE** statement also supports nested method calls by allowing one method call to be expressed as the target or the argument of another method. In these types of expressions, the result of the inner method is used to determine that of the outer method.

Partitioning

What is Partitioning?

Partitioning is a method by which a table is divided into multiple independent logical units. Each logical unit used in partitioning is called a partition. Partitioning can enhance manageability, performance and availability. Some advantages of partitioning are as follows:

- Improved management of large capacity tables
- Improved performance by narrowing the range of access when retrieving data
- Improved performance and decreased physical loads by distributing disk I/O
- Decreased possibility of data corruption and improved availability by partitioning a table into multiple chunks
- Optimized storage cost

Three types of partitioning methods are supported by CUBRID: range partitioning, hash partitioning, and list partitioning.

The maximum number of partitions cannot exceed 1,024. Each partition of a table is created as its subtable. The subtables created by the partitioning process cannot be altered or deleted by users. The name of the subtable is stored in the system table in a 'class_name__p__partition_name' format. Database users can check the partitioning information in the db_class and db_partition virtual tables. They can also check the information by using the ;sc <table name> command in the CUBRID Manager or the CSQL Interpreter.

Range Partitioning

Range Partitioning Definition

Description

You can define a range partition by using the **PARTITION BY RANGE** clause.

Syntax

```
CREATE TABLE(
...
)
PARTITION BY RANGE ( <partition_expression> ) (
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ),
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ) ),
... )
)
```

- *partition_expression* : Specifies the partition expression. The expression can be specified by the name of the column to be partitioned or by a function. For more information of the data types and functions available, see Data Types Available for Partition Expression.
- *partition_name* : Specifies the partition name.
- *range_value* : Specifies the partition-by value.

Example 1

The following is an example of creating the participant2 table with the participating countries, and inserting data that partitions the years into before and after the 2000 Olympic Games. When inserting data, the countries that participated in the 1988 and 1996 Olympic Games are stored in before_2000; the rest of them are stored in before_2008.

```
CREATE TABLE participant2 (host_year INT, nation CHAR(3), gold INT, silver INT, bronze INT)
PARTITION BY RANGE (host_year)
(PARTITION before_2000 VALUES LESS THAN (2000),
PARTITION before_2008 VALUES LESS THAN (2008) );

INSERT INTO participant2 VALUES (1988, 'NZL', 3, 2, 8);
```

```
INSERT INTO participant2 VALUES (1988, 'CAN', 3, 2, 5);
INSERT INTO participant2 VALUES (1996, 'KOR', 7, 15, 5);
INSERT INTO participant2 VALUES (2000, 'RUS', 32, 28, 28);
INSERT INTO participant2 VALUES (2004, 'JPN', 16, 9, 12);
```

Example 2

As shown below, the partition key value in a range partition is **NULL**, the data are stored in the first partition.

```
INSERT INTO participant2 VALUES(NULL, 'AAA', 0, 0, 0);
```

Caution

- The maximum number of partitions possible for a given table is 1024.
- If the partition key value is **NULL**, the data is stored in the first partition (see Example 2).

Range Partitioning Redefinition

Description

You can redefine a partition by using the **REORGANIZE PARTITION** clause of the **ALTER** statement. By redefining partitions, you can combine multiple partitions into one or divide one into multiple.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
REORGANIZE PARTITION
<alter partition name comma list>
INTO ( <partition definition comma list> )

partition definition comma list:
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ),....
```

- *table_name* : Specifies the name of the table to be redefined.
- *alter partition name comma list* : Specifies the partition to be redefined. Multiple partitions are separated by commas (,).
- *partition definition comma list* : Specifies the redefined partitions. Multiple partitions are separated by commas (,).

Example 1

The following is an example of repartitioning the before_2000 partition into the before_1996 and before_2000 partitions.

```
CREATE TABLE participant2 ( host_year INT, nation CHAR(3), gold INT, silver INT, bronze
INT)
PARTITION BY RANGE (host_year)
( PARTITION before_2000 VALUES LESS THAN (2000),
PARTITION before_2008 VALUES LESS THAN (2008) );

ALTER TABLE participant2 REORGANIZE PARTITION before_2000 INTO (
PARTITION before_1996 VALUES LESS THAN (1996),
PARTITION before_2000 VALUES LESS THAN (2000)
);
```

Example 2

The following is an example of combining two partitions redefined in Example 1 back into a single before_2000 partition.

```
ALTER TABLE participant2 REORGANIZE PARTITION before_1996, before_2000 INTO
(PARTITION before_2000 VALUES LESS THAN (2000) );
```

Caution

- When redefining a range or list partition, duplicate ranges or values are not allowed.
- The **REORGANIZE PARTITION** clause cannot be used to change the partition table type. For example, a range partition cannot be changed to a hash partition, or vice versa.

- The maximum number of partitions cannot exceed 1,024. There must be at least one partition remaining after deleting partitions. In a range-partitioned table, only adjacent partitions can be redefined.

Adding Range Partitioning

Description

You can add range partitions by using the **ADD PARTITION** clause of the **ALTER** statement.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
ADD PARTITION <partition definitions comma list>
partition definition comma list:
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ),...
```

- *table_name* : Specifies the name of the table to which partitions are added.
- *partition definition comma list* : Specifies the partitions to be added. Multiple partitions are separated by commas (,).

Example

Currently, the partition before the 2008 Olympic Games is defined in the participant2 table. The following is an example of adding the before_2012 and before_2016 partitions; the former will store the information about the 2012 Olympic Games and the latter will store the information about the 2016 Olympic Games.

```
ALTER TABLE participant2 ADD PARTITION (
PARTITION before_2012 VALUES LESS THAN (2012),
PARTITION before_2016 VALUES LESS THAN MAXVALUE );
```

Caution

- When a range partition is added, only the partition by value greater than the existing partition value can be added. Therefore, as shown in the above example, if the maximum value is specified by **MAXVALUE**, no more partitions can be added (you can add partitions by changing the **MAXVALUE** value by redefining the partition).
- To add the partition by value smaller than the existing partition value, use the redefining partitions (see Range Partitioning Redefinition).

Dropping Range Partitioning

Description

You can drop a partition by using the **DROP PARTITION** clause of the **ALTER** statement.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
DROP PARTITION <partition_name>
```

- *table_name* : Specifies the name of the partitioned table.
- *partition_name* : Specifies the name of the partition to be dropped.

Example

The following is an example of dropping the before_2000 partition in the participant2 table.

```
ALTER TABLE participant2 DROP PARTITION before_2000;
```

Caution

- When dropping a partitioned table, all stored data in the partition are also dropped.
- If you want to change the partitioning of a table without losing data, use the **ALTER TABLE**...**REORGANIZE PARTITION** statement (see Range Partitioning Redefinition).
- The number of rows deleted is not returned when a partition is dropped. If you want to delete the data, but want to maintain the table and partitions, use the **DELETE** statement.

Hash Partitioning

Hash Partitioning Definition

Description

You can define a hash partition by using the **PARTITION BY HASH** clause.

Syntax

```
CREATE TABLE (
...
)
( PATITION BY HASH ( <partition_expression> )
 PATITIONS ( <number_of_partitions> )
)
```

- *partition_expression* : Specifies a partition expression. The expression can be specified by the name of the column to be partitioned or by a function.
- *number_of_partitions* : Specifies the number of partitions.

Example 1

The following is an example of creating the nation2 table with country codes and country names, and defining 4 hash partitions based on code values. Only the number of partitions, not the name, is defined in hash partitioning; names such as p0 and p1 are assigned automatically.

```
CREATE TABLE nation2
( code CHAR(3),
name VARCHAR(50) )
PARTITION BY HASH ( code) PARTITIONS 4;
```

Example 2

The following is an example of inserting data to the hash partition created in the example 1. When a value is inserted into a hash partition, the partition to store the data is determined by the hash value of the partition key. If the partition key value is **NULL**, the data is stored in the first partition.

```
INSERT INTO nation2 VALUES ('KOR','Korea');
INSERT INTO nation2 VALUES ('USA','USA United States of America');
INSERT INTO nation2 VALUES ('FRA','France');
INSERT INTO nation2 VALUES ('DEN','Denmark');
INSERT INTO nation2 VALUES ('CHN','China');
INSERT INTO nation2 VALUES (NULL,'AAA');
```

Caution

- The maximum number of partitions cannot exceed 1,024.

Hash Partitioning Redefinition

Description

You can redefine a partition by using the **COALESCE PARTITION** clause of the **ALTER** statement. Instances are preserved if the hash partition is redefined.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
COALESCE PARTITION <unsigned integer>
```

- *table_name* : Specifies the name of the table to be redefined.
- *unsigned integer* : Specifies the number of partitions to be deleted.

Example

The following is an example of decreasing the number of partitions in the nation2 table from 4 to 2.

```
ALTER TABLE nation2 COALESCE PARTITION 2;
```

Caution

- Decreasing the number of partitions is only available.
- To increase the number of partitions, use the **ALTER TABLE** ... **ADD PARTITION** statement as in range partitioning (see Adding Range Partitioning for more information).
- There must be at least one partition remaining after redefining partitions.

List Partitioning

List Partitioning Definition

Description

You can define a list partition by using the **PARTITION BY LIST** statement.

Syntax

```
CREATE TABLE(
...
)
PARTITION BY LIST ( <partition_expression> ) (
PARTITION <partition_name> VALUES IN ( <partition_value_list> ),
PARTITION <partition_name> VALUES IN ( <partition_value_ list>
,
...
);
```

- *partition_expression* : Specifies a partition expression. The expression can be specified by the name of the column to be partitioned or by a function. For more information on the data types and functions available, see Data Types Available for Partition Expression.
- *partition_name* : Specifies the partition name.
- *partition_value_list* : Specifies the list of the partition by values.

Example 1

The following is an example of creating the athlete2 table with athlete names and sport events, and defining list partitions based on event values.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);
```

Example 2

The following is an example of inserting data to the list partition created in the example 1. In the last query of the example 2, if you insert an argument that has not been specified in the partition expression of the example 1, data inserting fails.

```
INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');
INSERT INTO athlete2 VALUES ('Moon Dae-Sung','Taekwondo');
INSERT INTO athlete2 VALUES ('Cho In-Chul', 'Judo');
INSERT INTO athlete2 VALUES ('Hong Kil-Dong', 'Volleyball');
```

Example 3

The following is an example where an error occurs with no data inserted when the partition key value is **NULL**. To define a partition where a **NULL** value can be inserted, define one that has a list including a **NULL** value as in the event3 partition as below.

```
INSERT INTO athlete2 VALUES ('Hong Kil-Dong','NULL');

CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball', NULL)
);
```

Caution

- The maximum number of partitions cannot exceed 1,024.

List Partitioning Redefinition

Description

You can redefine a partition by using the **REORGANIZE PARTITION** clause of the **ALTER** statement. By redefining partitions, you can combine multiple partitions into one or divide one into multiple.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
REORGANIZEPARTITION
<alter partition name comma list>
INTO ( <partition definition comma list> )
partition definition comma list:
PARTITION <partition_name> VALUES IN ( <partition_value_list>),...
```

- *table_name* : Specifies the name of the table to be redefined.
- *alter partition name comma list* : Specifies the partition to be redefined. Multiple partitions are separated by commas (,).
- *partition definition comma list* : Specifies the redefined partitions. Multiple partitions are separated by commas (,).

Example 1

The following is an example of creating the athlete2 table partitioned by the list of sport events, and redefining the event2 partition to be divided into event2_1 (Judo) and event2_2 (Taekwondo, Boxing).

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

ALTER TABLE athlete2 REORGANIZE PARTITION event2 INTO
(PARTITION event2_1 VALUES IN ('Judo'),
PARTITION event2_2 VALUES IN ( 'Taekwondo','Boxing'));
```

Example 2

The following is an example that combining the event2_1 and event2_2 partitions divided in Example 1 back into a single event2 partition.

```
ALTER TABLE athlete2 REORGANIZE PARTITION event2_1, event2_2 INTO
(PARTITION event2 VALUES IN('Judo','Taekwondo','Boxing'));
```

Dropping List Partitioning

Description

You can drop a partition by using the **DROP PARTITION** clause of the **ALTER** statement.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
DROP PARTITION <partition_name>
```

- *table_name* : Specifies the name of the partitioned table.
- *partition_name* : Specifies the name of the partition to be dropped.

Example

The following is an example of creating the athlete2 table partitioned by the list of sport events, and dropping the event3 partition.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

ALTER TABLE athlete2 DROP PARTITION event3;
```

Partitioning Management

Retrieving and Manipulating Data in Partitioning

Description

When retrieving data, the **SELECT** statement can be used not only for partitioned tables but also for each partition.

Example

The following is an example of creating the athlete2 table to be partitioned by the list of sport events, inserting data, and retrieving the event1 and event2 partitions.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');
INSERT INTO athlete2 VALUES ('Moon Dae-Sung','Taekwondo');
INSERT INTO athlete2 VALUES ('Cho In-Chul', 'Judo');
SELECT * from athlete2__p__event1;
  name                    event
============================================
  'Hwang Young-Cho'      'Athletics'

SELECT * from athlete2__p__event2;
  name                    event
============================================
  'Moon Dae-Sung'        'Taekwondo'
  'Cho In-Chul'          'Judo'
```

Caution

- Data manipulation such as insert, update and delete for each partition of the partitioned table is not allowed.

Moving Data by Changing Partitioning Key Value

Description

If a partition key value is changed, the changed instance can be moved to another partition by the partition expression.

Example

The following is an example of moving the instance to another partition by changing the partition key value.

If you change the sport event information of Hwang Young-Cho in the event1 partition from Athletics to Football, the instance is moved to the event3 partition.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');

  name                  event
============================================
  'Hwang Young-Cho'     'Athletics'

UPDATE athlete2 SET event = 'Football' WHERE name = 'Hwang Young-Cho';

SELECT * FROM athlete2__p__event3;
  name                  event
============================================
  'Lee Seung-Yuop'      'Baseball'
  'Hwang Young-Cho'     'Football'
```

Caution

- Be aware that when moving data between partitions by changing a partition key value, it can cause performance degradation due to internal deletions and insertions.

Altering Regular Table into Partitioning Table

Description

To alter a regular table into a partitioned one, use the **ALTER TABLE** statement. Three partitioning methods can be used with the **ALTER TABLE** statement. The data in the existing table are moved to and stored in each partition according to the partition definition.

Syntax

```
ALTER {TABLE | CLASS} table_name
PARTITION BY {RANGE | HASH | LIST } ( <partition_expression> )
( PARTITION partition_name VALUES LESS THAN { MAXVALUE | ( <partition_value_option> ) }
| PARTITION partition_name VALUES IN ( <partition_value_option list) > ]
| PARTITION <UNSINGED_INTEGER> )

<partition_expression>
expression_
<partition_value_option>
literal_
```

- *table_name* : Specifies the name of the table to be altered.
- *partition_expression* : Specifies a partition expression. The expression can be specified by the name of the column to be partitioned or by a function. For more information on the data types and functions available, see Data Types Available for Partition Expressions.
- *partition_name* : Specifies the name of the partition.
- *partition_value_option* : Specifies the value or the value list on which the partition is based.

Example

The following are examples of altering the record table into a range, list and hash table respectively.

```
ALTER TABLE record PARTITION BY RANGE (host_year)
( PARTITION before_1996 VALUES LESS THAN (1996),
  PARTITION after_1996 VALUES LESS THAN MAXVALUE);

ALTER TABLE record PARTITION BY list (unit)
( PARTITION time_record VALUES IN ('Time'),
  PARTITION kg_record VALUES IN ('kg'),
  PARTITION meter_record VALUES IN ('Meter'),
  PARTITION score_record VALUES IN ('Score') );

ALTER TABLE record
PARTITION BY HASH (score) PARTITIONS 4;
```

Caution

- If there is data that does not satisfy the partition condition, partitions cannot be defined.

Altering Partitioning Table into Regular Table

Description

To alter an existing partitioned table into a regular one, use the **ALTER TABLE** statement.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
REMOVE PARTITIONING
```

- *table_name* : Specifies the name of the table to be altered.

Example

The following is an example of altering the partitioned table of name "nation2" into a regular one.

```
ALTER TABLE nation2 REMOVE PARTITIONING;
```

Partition Pruning

Description

Partition pruning is an optimization, limiting the scope of your query according to the criteria you have specified. It is the skipping of unnecessary data partitions in a query. By doing this, you can greatly reduce the amount of data output from the disk and time spent on processing data as well as improve query performance and resource availability.

Example 1

The following is an example of creating the olympic2 table to be partitioned based on the year the Olympic Games were held, and retrieving the countries that participated in the Olympic Games since the 2000 Sydney Olympic Games.

In the **WHERE** clause, partition pruning takes place when equality or range comparison is performed between a partition key and a constant value. In this example, the before_1996 partition that has a smaller year value than 2000 is not scanned.

```
CREATE TABLE olympic2
( opening_date DATE, host_nation VARCHAR(40))
PARTITION BY RANGE ( EXTRACT (YEAR FROM opening_date) )
( PARTITION before_1996 VALUES LESS THAN (1996),
  PARTITION before_MAX VALUES LESS THAN MAXVALUE );

SELECT opening_date, host_nation FROM olympic2 WHERE EXTRACT ( YEAR FROM (opening_date))
>= 2000;
```

Example 2

The following is an example of showing the method of getting the effects of partition pruning by retrieving data with a specific partition when partition pruning does not occur. In the first query, partition pruning does not occur because the value compared is not in the same format as that of the partition expression.

Therefore, you can use the same effect of partition pruning by specifying the appropriate partition as shown in the second query.

```
SELECT host_nation FROM olympic2 WHERE opening_date >= '2000 - 01 - 01';

SELECT host_nation FROM olympic2__p__before_max WHERE opening_date >= '2000 - 01 - 01';
```

Example 3

The following is an example of specifying the search condition to make a partition pruning in the hash partitioned table, called the manager table.

For hash partitioning, partition pruning occurs only when equality comparison is performed between a partition key and a constant value in the **WHERE** clause.

```
CREATE TABLE manager (
code INT,
name VARCHAR(50))
PARTITION BY HASH ( code) PARTITIONS 4;

SELECT * FROM manager WHERE code = 10053;
```

Caution

- The partition expression and the value compared must be in the same format.

Data Types Available for Partitioning Expression

Description

The following table shows data types of the column that can or cannot be used as a partition key.

Data Types Available	Data Types Unavailable
CHAR	FLOAT
VARCHAR	REAL
NCHAR	DOUBLE
VARNCHAR	BIT
INTEGER	BIT VARYING
SMALLINT	NUMERIC OR DECIMAL
DATE	MONETARY
TIME	SET
TIMESTAMP	LIST OR SEQUENCE
	MULTISET
	OBJECT

The following operator functions can be used in partition expressions to be applied to partition keys.

- Number Operations

  +, -, *, /, MOD, STRCAT, FLOOR, CEIL, POWER, ROUND, ABS, TRUNC

- String Operations

  POSITION, SUBSTRING, OCTEC_LENGTH, BIT_LENGTH, CHAR_LENGTH, LOWER, UPPER, TRIM, LTRIM, RTRIM, LPAD, RPAD, REPLACE, TRANSLATE

- Date Operations

  ADD_MONTH, LAST_DAY, MONTH_BETWEEN, SYS_DATE, SYS_TIME, SYS_TIMESTAMP, TO_DATE, TO_NUMBER, TO_TIME, TO_TIMESTAMP, TO_CHAR

- Others

EXTRACT, CAST

Creating VIEW with Partitioning Table

Description

You can define a virtual table by using each partition of a partitioned table. Retrieving data from the virtual table created is possible, but data insert, delete and update operations are not allowed.

Example

The following is an example of creating the participant2 table partitioned based on the participating year, and creating and retrieving a virtual table with the participant2__p__before_2000 partition.

```
CREATE TABLE participant2 (host_year INT, nation CHAR(3), gold INT, silver INT, bronze INT)
PARTITION BY RANGE (host_year)
( PARTITION before_2000 VALUES LESS THAN (2000),
 PARTITION before_2008 VALUES LESS THAN (2008) );

INSERT INTO participant2 VALUES (1988, 'NZL', 3, 2, 8);
INSERT INTO participant2 VALUES (1988, 'CAN', 3, 2, 5);
INSERT INTO participant2 VALUES (1996, 'KOR', 7, 15, 5);
INSERT INTO participant2 VALUES (2000, 'RUS', 32, 28, 28);
INSERT INTO participant2 VALUES (2004, 'JPN', 16, 9, 12);

CREATE VIEW v_2000 AS
SELECT * FROM participant2__p__before_2000
WHERE host_year = 1988;
    host_year  nation                       gold       silver       bronze
==========================================================================
         1988  'NZL'                           3            2            8
         1988  'CAN'                           3            2            5
```

Updating Statistics on Partitioning Tables

You can update statistics on the database by using the **cubrid optimizedb** utility or the SQL statement called **UPDATE STATISTICS ON CLASSES**. You can also use the **ANALYZE PARTITION** statement for partitioned tables.

The following is an example of the **ANALYZE PARTITION** statement.

```
ALTER TABLE t1 ANALYZE PARTITION p3;
```

Class Inheritance

Overview

Description

To explain the concept of inheritance, a table is represented as a class and a column is represented as an attribute.

Classes in CUBRID database can have class hierarchy. Attributes and methods can be inherited through such hierarchy.

As shown in the previous section, you can create a Manager class by inheriting attributes from an Employee class. The Manager class is called the **subclass** of the Employee class, and the Employee class is called the **super class** of the Manager class. Inheritance can simplify class creation by reusing the existing class hierarchy.

CUBRID allows multiple inheritance, which means that a class can inherit attributes and methods from more than one super class. However, inheritance can cause conflicts when an attribute or method of the super class is added or deleted.

Such conflict occurs in multiple inheritance if there are attributes or methods with the same name in different super classes. For example, if it is likely that a class inherits attributes of the same name and type from more than one super class, you must specify the attributes to be inherited. In such a case, if the inherited super class is deleted, a new attribute of the same name and type must be inherited from another super class. In most cases, the database system resolves such problems automatically. However, if you don't like the way that the system resolves a problem, you can resolve it manually by using the INHERIT clause.

When attributes are inherited from more than one super class, it is possible that their names are to be the same, while their domains are different. For example, two super classes may have the same attribute, whose domain is a class. In this case, a subclass automatically inherits attributes with more specialized (a lower in the class hierarchy) domains. If such conflict occurs between basic data types (e.g. STRING or INTEGER) provided by the system, inheritance fails. Conflicts during inheritance and their resolutions will be covered in the Resolving Class Conflicts section.

Caution

The following cautions must be observed during inheritance:

- The class name must be unique in the database. A class can be created as a subclass of one or more super class names in the database optionally. An error occurs if you create a class that inherits another class that does not exist.
- The name of a method/attribute must be unique within a class. The name cannot contain spaces, and cannot be a reserved keyword of CUBRID. Alphabets as well as '_', '#', '%' are allowed in the class name, but the first character cannot be '_'. A class name cannot exceed 255 English letters. Class names are not case-sensitive. A class name will be saved in the system after being converted to lowercase characters.

Note A super class name can begin with the user name so that the owner of the class can be easily identified.

Class Attribute and Method

You can create class attributes to store the aggregate property of all instances in the class. When you define a **CLASS** attribute or method, you must precede the attribute or method name with the keyword **CLASS**. Because a class attribute is associated with the class itself, not with an instances of the class, it has only one value. For example, a class attribute can be used to store the average value determined by a class method or the timestamp when the class was created. A class method is executed on the class object itself. It can be used to calculate the aggregate value for the instances of the class.

When a subclass inherits a super class, each class has a separate storage space for class attributes, so that two classes may have different values of class attribute. Therefore, the subclass does not change even when the attributes of the super class are changed.

The name of a class attribute can be the same as that of an instance attribute of the same class. Likewise, the name of a class method can be the same as that of an instance method of the same class.

Order Rule for Inheritance

The following rules apply to inheritance. The term class is generally used to describe the inheritance relationship between classes and virtual classes in the database.

- For an object without a super class, attributes are defined in the same order as in the **CREATE** statement (an ANSI standard).
- If there is one super class, locally created attributes are placed after the super class attributes. The order of the attributes inherited from the super class follows the one defined during the super class definition. For multiple inheritance, the order of the super class attributes is determined by the order of the super classes specified during the class definition.
- If more than one super class inherits the same class, the attribute that exists in both super classes is inherited to the subclass only once. At this time, if a conflict occurs, the attribute of the first super class is inherited.
- If a name conflict occurs in more than one super class, you can inherit only the ones you want from the super class attributes by using the **INHERIT** clause in order to resolve the conflict.
- If the name of the super class attribute is changed by the alias option of the **INHERIT** clause, its position is maintained.

INHERIT Clause

Description

When a class is created as a subclass, the class inherits all attributes and methods of the super class. A name conflict that occurs during inheritance can be handled by either a system or a user. To resolve the name conflict directly, add the **INHERIT** clause to the **CREATE CLASS** statement.

Syntax

```
CREATE CLASS
.
.
.
INHERIT resolution [ {, resolution }_ ]

resolution :
{ column_name | method_name } OF super class [ AS alias ]
```

For the *attr_mthd_name* in the **INHERIT** clause, specify the name of the attribute or method of the super class to inherit. With the **ALIAS** clause, you can resolve a name conflict that occurs in multiple inheritance statements by inheriting a new name.

ADD SUPERCLASS Clause

Description

To extend class inheritance, add a super class to a class. A relationship between two classes is created when a super class is added to an existing class. Adding a super class does not mean adding a new class.

Syntax

```
ALTER CLASS
.
.
.
ADD super class [ user_name.]class_name [ { , [ user_name.]class_name }_ ]
[ INHERIT resolution [ {, resolution }_ ] ] [ ]
resolution:
{ column_name | method_name } OF super class_name [ AS alias ]
```

For the first *class_name*, specify the name of the class where a super class is to be added. Attributes and methods of the super class can be inherited by using the syntax above.

Name conflicts can occur when adding a new super class. If a name conflict cannot be resolved by the database system, attributes or methods to inherit from the super class can be specified by using the **INHERIT** clause. You can use aliases to inherit all attributes or methods that cause the conflict. For more information on super class name conflicts, see the Resolving Class Conflict section.

Example

The following is an example of creating the female_event class by inheriting the event class included in demodb.

```
CREATE CLASS female_event UNDER event
```

DROP SUPERCLASS Clause

Description

Deleting a super class from a class means removing the relationship between two classes. If a super class is deleted from a class, it changes inheritance relationship of the classes as well as of all their subclasses.

Syntax

```
ALTER CLASS
.
.
.
DROP super class class_name [ { , class_name }_ ]
[ INHERIT resolution [ {, resolution }_ ] ] [ ]

resolution:
{ column_name | method_name } OF super class_name [ AS alias ]
```

For the first *class_name*, specify the name of the class to be modified. For the second *class_name*, specify the name of the super class to be deleted. If a name conflict occurs after deleting a super class, see the Resolving Class Conflict section for the resolution.

Example 1

In the following example, the female_event class inherits from the event class.

```
CREATE CLASS female_event UNDER event
```

Example 2

In the following example, the **ALTER** statement deletes the event super class from the female_event class. The attributes that the female_event class inherited from the event class do not exist any more.

```
ALTER CLASS female_event
DROP super class event
```

Class Conflict Resolution

Overview

If you modify the schema of the database, conflicts can occur between attributes or methods of inheritance classes. Most conflicts are resolved automatically by CUBRID otherwise, you must resolve the conflict manually. Therefore, you need to examine the possibility of conflicts before modifying the schema.

Two types of conflicts can cause damage to the database schema. One is conflict with a subclass when the subclass schema is modified. The other is conflict with a super class when the super class is modified. The following are operations that may cause conflicts between classes.

- Adding an attribute
- Deleting an attribute
- Adding a super class
- Deleting a super class
- Deleting a class

If a conflict occurs as the result of the above operations, CUBRID applies a basic resolution to the subclass where the conflict occurred. Therefore, the database schema can always maintain consistent state.

Resolution Specifier

Description

Conflicts between the existing classes or attributes, and inheritance conflicts can occur if the database schema is modified. If the system fails to resolve a conflict automatically or if you don't like the way the system resolved the problem, you can suggest how to resolve the conflict by using the **INHERIT** clause of the **ALTER** statement (often referred as resolution specifier).

When the system resolves the conflict automatically, basically, the existing inheritance is maintained (if any). If the previous resolution becomes invalid when the schema is modified, the system will arbitrarily select another one. Therefore, you must avoid excessive reuse of attributes or methods in the schema design stage because the way the system will resolve the conflict cannot always be predictable.

What will be discussed concerning conflicts is applied commonly to both attributes and methods.

Syntax

```
ALTER [ class_type ] class_name alter_clause
[ INHERIT resolution [ {, resolution }_ ] ] [ ]
resolution:
{ column_name | method_name } OF super class_name [ AS alias ]
```

Superclass Conflict

Adding a super class

The **INHERIT** clause of the **ALTER CLASS** statement is optional, but must be used when a conflict occurs due to class changes. You can specify more than one resolutions after the **INHERIT** clause.

super class_name specifies the name of the super class that has the new attribute or method to inherit when a conflict occurs. *attr_mthd_name* specifies the name of the attribute or method to inherit. You can use the **alias** clause when you need to change the name of the attribute or method to inherit.

The following example creates the soccer_stadium class by inheriting the event and stadium classes in the olympic database of demodb. Because both event and stadium classes have the name and code attributes, you must specify the attributes to inherit using the **INHERIT** clause.

```
CREATE CLASS soccer_stadium UNDER event, stadium
INHERIT name OF stadium, code OF stadium
```

When the two super classes (event and stadium) have the name attribute, if the soccer_stadium class needs to inherit both attributes, it can inherit the name unchanged from the stadium class and the name changed from the event class by using the **alias** clause of the **INHERIT**.

The following is an example in which the name attribute of the stadium class is inherited as it is, and that of the event class is inherited as the 'purpose' alias.

```
ALTER CLASS soccer_stadium
INHERIT name OF event AS purpose
```

Deleting a super class

A name conflict may occur again if a super class that explicitly inherited an attribute or method is dropped by using the **INHERIT**. In this case, you must specify the attribute or method to be explicitly inherited when dropping the super class.

The following is an example of creating the seoul_1988_soccer class by inheriting game, participant and stadium classes from demodb, and deleting the participant class from the super class. Because nation_code and host_year are explicitly inherited from the participant class, you must resolve their name conflicts before deleting it from the super class. However, host_year does not need to be specified explicitly because it exists only in the game class.

```
CREATE CLASS seoul_1988_soccer UNDER game, participant, stadium
INHERIT nation_code OF participant, host_year OF participant
ALTER CLASS seoul_1988_soccer
DROP super class participant
INHERIT nation_code OF stadium
```

Compatible Domains

When an attribute conflict occurs among two or more super classes, the statement resolving the conflict is not possible only if all attributes have compatible domains.

For example, the class that inherits a super class with the phone attribute of integer type cannot have another super class with the phone attribute of string type. If the types of the phone attributes of the two super classes are both String or Integer, you can add a new super class by resolving the conflict with the **INHERIT** clause.

Compatibility is checked when inheriting an attribute with the same name, but with the different domain. In this case, the attribute that has a lower class in the class inheritance hierarchy as the domain is automatically inherited. If the domains of the attributes to inherit are compatible, the conflict must be resolved in the class where an inheritance relationship is defined.

Subclass Conflict

Any changes in a class will be automatically propagated to all subclasses. If a problem occurs in the subclass due to the changes, CUBRID resolves the corresponding subclass conflict and then displays a message saying that the conflict has been resolved automatically by the system.

Subclass conflicts can occur due to operations such as adding a super class, or creating/deleting a method or an attribute. Any changes in a class will affect all subclasses. Since changes are automatically propagated, harmless changes can even cause side effects in subclasses.

Adding Attributes and Methods

The simplest subclass conflict occurs when an attribute is added. A subclass conflict occurs if an attribute added to a super class has the same name as one already inherited by another super class. In such cases, CUBRID will automatically resolve the problem. That is, the added attribute will not be inherited to all subclasses that have already inherited the attribute with the same name.

The following is an example of adding an attribute to the event class. The super classes of the soccer_stadium class are the event and the stadium classes, and the nation_code attribute already exists in the stadium class. Therefore, a conflict occurs in the soccer_stadium class if the nation_code attribute is added to the event class. However, CUBRID resolves this conflict automatically.

```
ALTER CLASS event
ADD ATTRIBUTE nation_code CHAR(3)
```

If the event class is dropped from the soccer_stadium super class, the cost attribute of the stadium class will be inherited automatically.

Dropping Attributes and Methods

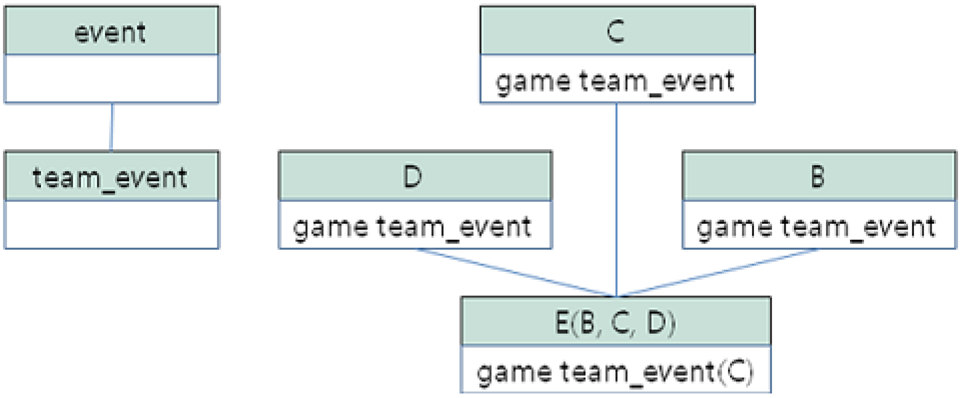
When an attribute is dropped from a class, any resolution specifiers which refer to the attribute by using the **INHERIT** clause are also removed. If a conflict occurs due to the deletion of an attribute, the system will determine a new inheritance hierarchy. If you don't like the inheritance hierarchy determined by the system, you can determine it by using the **INHERIT** clause of the **ALTER** statement. The following is an example of such conflict.

Suppose there is a subclass that inherits attributes from three different super classes. If a name conflict occurrs in all super classes and the explicitly inherited attribute is dropped, one of the remaining two attributes will be inherited automatically to resolve the problem.

The following is an example of a subclass conflict. Classes B, C and D are super classes of class E, and have an attribute whose name is team and the domain is team_event. Class E was created with the place attribute inherited from class C as follows:

```
create class E under B, C, D
inherit place of C
```

In this case, the inheritance hierarchy is as follows:



Suppose that you decide to delete class C from the super class. This drop will require changes to the inheritance hierarchy. Because the domains of the remaining classes B and D with the game attribute are at the same level, the system will randomly choose to inherit from one of the two classes. If you don't want the system to make a random selection, you can specify the class to inherit from by using the **INHERIT** clause when you change the class.

```
ALTER CLASS E
INHERIT game OF D
ALTER CLASS C
DROP game
```

Note If the domain of the game attribute of one super class is event and that of another super class is team_event, the attribute that has team_event as the domain will be inherited because team_event is more specific than event (as team_event exists lower in the inheritance hierarchy). In this case, you cannot force the attribute that has event as the domain to be inherited because the event class exists higher in the inheritance hierarchy than team_event.

Schema Invariant

Invariants of a database schema are a property of the schema that must be preserved consistently (before and after the schema change). There are four types of invariants: invariants of class hierarchy, name, inheritance and consistency.

- **Invariant of class hierarchy** has a single root and defines a class hierarchy as a Directed Acyclic Graph (DAG) where all connected classes have a single direction. That is, all classes except for the root have one or more super classes, and cannot become their own super classes. The root of DAG is "object," a system-defined class.
- **Invariant of name** means that all classes in the class hierarchy and all attributes in a class must have unique names. That is, attempts to create classes with the same name or to create attributes or methods with the same name in a single class are not allowed.

  Invariant of name is redefined by the 'rename' qualifier. The 'rename' qualifier allows the name of an attribute or method to be changed.
- **Invariant of inheritance** means that a class must inherit all attributes and methods from all super classes. This invariant can be distinguished with three qualifiers: source, conflict and domain. The names of inherited attributes and methods can be modified. For default or shared value attributes, the default or shared value can be modified. Invariant of inheritance means that such changes will be propagated to all classes that inherit these attributes and methods.
- A **source qualifier** means that if class C inherits subclasses of class S, only one of the subclass attributes (methods) inherited from class S can be inherited to class C. That is, if an attribute (method) defined in class S is inherited by other classes, it is in effect a single attribute (method), even though it exists in many subclasses. Therefore, if a class multiply inherits from classes that have attributes (methods) of the same source, only one appearance of the attribute (method) is inherited.
- A **conflict qualifier** means that if class C inherits from two or more classes that have attributes (methods) with the same name but of different sources, it can inherit more than one class. To inherit attributes (methods) with the same name, you must change their names so as not to violate the invariant of name.
- A **domain qualifier** means that a domain of an inherited attribute can be converted to the domain's subclass.
- **Invariant of consistency** means that the database schema must always follow the invariants of a schema and all rules (Rules for Schema Changes) except when it is being changed.

Rule for Schema Changes

The Invariants of a Schema section has described the characteristics of schema that must be preserved all the time. There are some methods for changing schemas, and all these methods must be able to preserve the invariants of a schema. For example, suppose that in a class which has a single super class, the relationship with the super class is to be removed. If the relationship with the super class is removed, the class becomes a direct subclass of the object class, or the removal attempt will be rejected if the user specified that the class should have at least one super class. To have some rules for selecting one of the methods for changing schemas, even though such selection seems arbitrary, will be definitely useful to users and database designers.

The following three types of rules apply: conflict-resolution rules, domain-change rule and class-hierarchy rule.

Seven conflict-resolution rules reinforce the invariant of inheritance. Most schema change rules are needed because of name conflicts. A domain-change rule reinforces a domain resolution of the invariant of inheritance. A class-hierarchy rule reinforces the invariant of class hierarchy.

Conflict-Resolution Rules

- **Rule 1** : If an attribute (method) name of class C and an attribute name of the super class S conflict with each other (that is, their names are same), the attribute of class C is used. The attribute of S is not inherited.

  If a class has one or more super classes, three aspects of the attribute (method) of each super class must be considered to determine whether the attributes are semantically equal and which attribute to inherit. The three aspects of the attribute (method) are the name, domain and source. The following table shows eight combinations of these three aspects that can happen with two super classes. In Case 1 (two different super classes have attributes with the same name, domain and source), only one of the two subclasses should be inherited because two attributes are identical. In Case 8 (two different super classes have attributes with different names, domains and sources), both classes should be inherited because two attributes are totally different ones.

Case	Name	Domain	Source
1	Same	Same	Same
2	Same	Same	Different
3	Same	Different	Same

4	Same	Different	Different
5	Different	Same	Same
6	Different	Same	Different
7	Different	Different	Same
8	Different	Different	Different

Five cases (1, 5, 6, 7, 8) out of eight have clear meaning. Invariant of inheritance is a guideline for resolving conflicts in such cases. In other cases (2, 3, 4), it is very difficult to resolve conflicts automatically. Rules 2 and 3 can be resolutions for these conflicts.

- **Rule 2** : When two or more super classes have attributes (methods) with different sources but the same name and domain, one or more attributes (methods) can be inherited if the conflict-resolution statement is used. If the conflict-resolution statement is not used, the system will select and inherit one of the two attributes.

  This rule is a guideline for resolving conflicts of Case 2 in the table above.

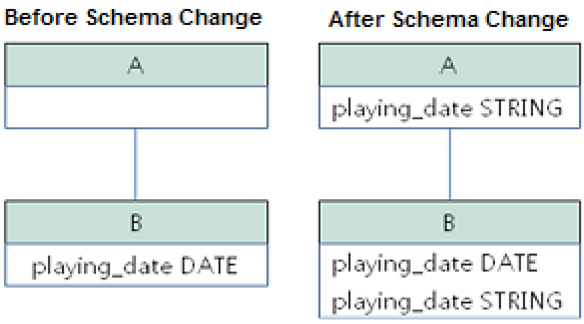
- **Rule 3** : If two or more super classes have attributes with different sources and domains but the same name, attributes (methods) with more detailed (lower in the inheritance hierarchy) domains are inherited. If there is no inheritance relationship between domains, schema change is not allowed.

  This rule is a guideline for resolving conflicts of Case 3 and 4. If Case 3 and 4 conflict with each other, Case 3 has the priority.

- **Rule 4** : The user can make any changes except for the ones in Case 3 and 4. In addition, the resolution of subclass conflicts cannot cause changes in the super class.

  The philosophy of Rule 4 is that "an inheritance is a privilege a subclass obtained from a super class, so changes in a subclass cannot affect the super class." Rule 4 means that the name of the attribute (method) included in the super class cannot be changed to resolve conflicts between class C and super classes. Rule 4 has an exception in cases where the schema change causes conflicts in Case 3 and 4.

- For example, suppose that class A is the super class of class B, and class B has the playing_date attribute of **DATE** type. If an attribute of **STRING** type named playing_date is added to class A, it conflicts with the playing_date attribute in class B. This is what happens in Case 4. The precise way to resolve such conflict is for the user to specify that class B must inherit the playing_date attribute of class A. If a method refers to the attribute, the user of class B needs to modify the method properly so that the appropriate playing_date attribute will be referenced. Schema change of class A is not allowed because the schema falls into an inconsistent state if the user of class B does not describe an explicit statement to resolve the conflict occurring from the schema change.



- **Rule 5** : If a conflict occurs due to a schema change of the super class, the original resolution is maintained as long as the change does not violate the rules. However, if the original resolution becomes invalid due to the schema change, the system will apply another resolution.

  Rule 5 is for cases where a conflict is caused to a conflict-free class or where the original resolution becomes invalid.

  This is the case where the name or domain of an attribute (method) is modified or a super class is deleted when the attribute (method) is added to the super class or the one inherited from the super class is deleted. The philosophy of Rule 5 coincides with that of Rule 4. That is, the user can change the class freely without considering what effects the subclass that inherits from the given class will have on the inherited attribute (method).

  When you change the schema of class C, if you decide to inherit an attribute of the class due to an earlier conflict with another class, this may cause attribute (method) loss of class C. Instead, you must inherit one of the attributes (methods) that caused conflicts earlier.

The schema change of the super class can cause a conflict between the attribute (method) of the super class and the (locally declared or inherited) attribute (method) of class C. In this case, the system resolves the conflict automatically by applying Rule 2 or 3 and may inform the user.

Rule 5 cannot be applied to cases where a new conflict occurs due to the addition or deletion of the relationship with the super class. The addition/deletion of a super class must be limited to within the class. That is, the user must provide an explicit resolution.

- **Rule 6** : Changes of attributes or methods are propagated only to subclasses without conflicts.

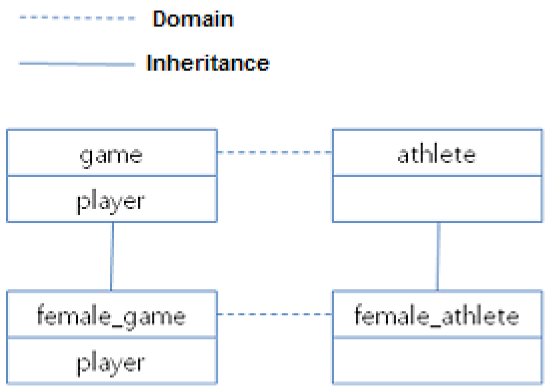
  This rule limits the application of Rule 5 and the invariant of inheritance. Conflicts can be detected and resolved by applying Rule 2 and 3.

- **Rule 7** : Class C can be dropped even when an attribute of class R uses class C as a domain. In this case, the domain of the attribute that uses class C as a domain can be changed to object.

Domain-Change Rule

- **Rule 8** : If the domain of an attribute of class C is changed from D to a super class of D, the new domain is less generic than the corresponding domain in the super class from which class C inherited the attribute. The following example explains the principle of this rule.

  Suppose that in the database there are the game class with the player attribute and the female_game class which inherits game. The domain of the player attribute of the game class is the athlete class, but the domain of the player attribute of the female_game class is changed to female_athlete which is a subclass of athlete. The following diagram shows such relationship. The domain of the player attribute of the female_game class can be changed back to athlete, which is the super class of female_athlete.



Class-Hierarchy Rule

- **Rule 9** : A class without a super class becomes a direct subclass of object. The class-hierarchy rule defines characteristics of classes without super classes. If you create a class without a super class, object becomes the super class. If you delete the super class S, which is a unique super class of class C, class C becomes a direct subclass of object.

CUBRID System Catalog

Overview

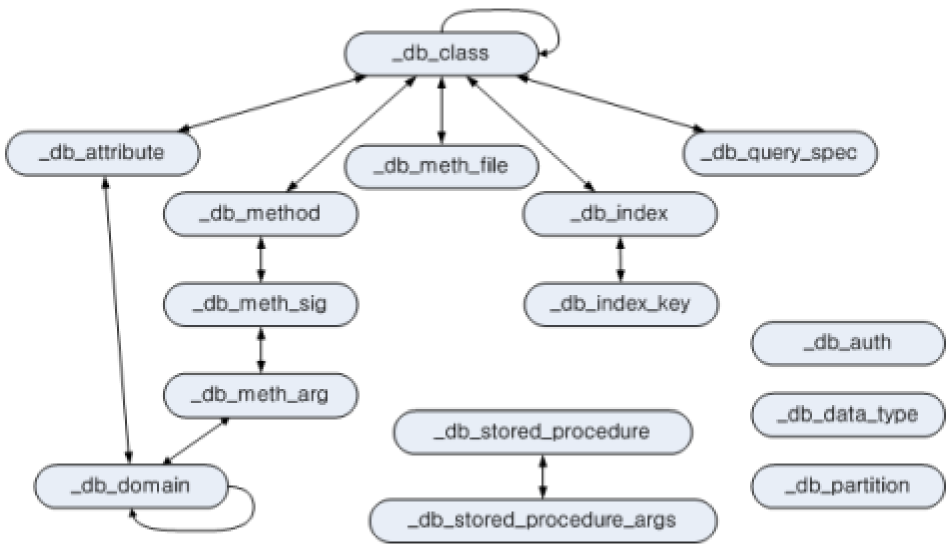
You can easily get various schema information from the SQL statement by using the system catalog virtual class (table). For example, you can get the following schema information by using the catalog virtual class.

```
-- Classes that refer to the 'b_user' class
SELECT class_name
FROM db_attribute
WHERE domain_class_name = 'db_user'

-- The number of classes that the current user can access
SELECT COUNT(*)
FROM db_class

-- Attribute of the 'db_user' class
SELECT attr_name, data_type
FROM db_attribute
WHERE class_name = 'db_user'
```

System Catalog Classes

System Catalog Classes

To define a catalog virtual class, define a catalog class first. The figure below shows catalog classes to be added and their relationships. The arrows represent the reference relationship between classes, and the classes that start with an underline (_) are catalog classes.



Added catalog classes represent information about all classes, attributes and methods in the database. Catalog classes are made up of class composition hierarchy and designed to have OIDs of catalog class instances for cross reference.

_db_class

Represents information about the class. An index for class_name is created.

Attribute Name	Data Type	Description
class_of	object	A class object. Represents a meta information object for the class saved in the system.
inst_attr_count	INTEGER	The number of instance attributes

shared_attr_count	INTEGER	The number of shared attributes
inst_meth_count	INTEGER	The number of instance methods
class_meth_count	INTEGER	The number of class methods
class_attr_count	INTEGER	The number of class attributes
is_system_class	INTEGER	0 for a user-defined class, and 1 for a system class.
class_type	INTEGER	0 for a class, and 1 for a virtual class.
owner	db_user	Class owner
class_name	VARCHAR(255)	Class name
sub_classes	SEQUENCE OF _db_class	Class one level down
super_classes	SEQUENCE OF _db_class	Class one level up
inst_attrs	SEQUENCE OF _db_attribute	Instance attribute
shared_attrs	SEQUENCE OF _db_attribute	Shared attribute
class_attrs	SEQUENCE OF _db_attribute	Class attribute
inst_meths	SEQUENCE OF _db_method	Instance method
class_meths	SEQUENCE OF _db_method	Class method
meth_files	SEQUENCE OF _db_methfile	File path in which the function for the method is located
query_specs	SEQUENCE OF _db_queryspec	SQL definition statement for a virtual class
indexes	SEQUENCE OF _db_index	Index created in the class

Example

The following is an example of retrieving all subclasses under the class owned by user 'PUBLIC' (for the child class female_event in the result, see the example in Adding a super class).

```
SELECT class_name, SEQUENCE(SELECT class_name FROM _db_class s WHERE s IN c.sub_classes)
 FROM _db_class c
 WHERE c.owner.name = 'PUBLIC' AND c.sub_classes IS NOT NULL;
  class_name            sequence((select class_name from _db_class s where s in
c.sub_classes))
============================================
  'event'               {'female_event'}
```

Note All examples of system catalog classes have been written in the csql utility. In this example, **--no-auto-commit** (inactive mode of auto-commit) and **-u** (specifying user DBA) options are used.

% csql --no-auto-commit -u dba demodb

_db_attribute

Represents information about attributes. Indexes for class_of and attr_name are created.

Attribute Name	Data Type	Description
class_of	_db_class	Class to which the attribute belongs

attr_type	INTEGER	Type defined for the attribute. 0 for an instance attribute, 1 for a class attribute, and 2 for a shared attribute.
data_type	INTEGER	Data type of the attribute. One of the values specified in the "Data Types Supported by CUBRID" table below.
def_order	INTEGER	Order of attributes in the class. Begins with 0. If the attribute is inherited, the order is the one defined in the super class. For example, if class y inherits attribute a from class x and a was first defined in x, def_order becomes 0.
from_class_of	_db_class	If the attribute is inherited, the super class in which the attribute is defined is used. Otherwise, **NULL**
from_attr_name	VARCHAR(255)	If the attribute is inherited and its name has been changed to resolve a name conflict, the original name defined in the super class is used. Otherwise, **NULL**
attr_name	VARCHAR(255)	Attribute name
default_value	VARCHAR(255)	Default value. Saved as a character string regardless of data types. If there is no default value, **NULL**. If the default value is **NULL**, 'NULL' is used. If the data type is an object, 'volume id | page id | slot id' is used. If the data type is a set, '{element 1, element 2, ... is used.
domains	SEQUENCE OF _db_domain	Domain information of the data type
is_nullable	INTEGER	0 if a not null constraint is configured, and 1 otherwise.

Data Types Supported by CUBRID

Value	**Meaning**	Value	Meaning
1	INTEGER	13	MONETARY
2	FLOAT	18	SHORT
3	DOUBLE	20	OID
4	STRING	22	NUMERIC
5	OBJECT	23	BIT
6	SET	24	VARBIT
7	MULTISET	25	CHAR
8	SEQUENCE	26	NCHAR
9	ELO	27	VARNCHAR
10	TIME	31	BIGINT
11	TIMESTAMP	32	DATETIME
12	DATE	33	BLOB
		34	CLOB

Character Sets Supported by CUBRID

Value	**Meaning**
0	US English ? ASCII encoding
3	Latin 1 ? ISO 8859 encoding
4	KSC 5601 1990 ? EUC encoding

Example

The following is an example of retrieving user classes (from_class_of.is_system_class = 0) among the ones owned by user 'PUBLIC.'

```
SELECT class_of.class_name, attr_name
FROM _db_attribute
WHERE class_of.owner.name = 'PUBLIC' AND FROM _class_of.is_system_class = 0
ORDER BY 1, def_order;
class_of.class_name   attr_name
============================================
  'female_event'        'code'
  'female_event'        'sports'
  'female_event'        'name'
  'female_event'        'gender'
  'female_event'        'players'
```

_db_domain

Represents information about the domain. An index for object_of is created.

Attribute Name	Data Type	Description
object_of	object	Attribute that refers to the domain, which can be a method parameter or domain
data_type	INTEGER	Data type of the domain (a value in the "Value" column of the "Data Types Supported by CUBRID" table in _db_attribute)
prec	INTEGER	Precision of the data type. 0 is used if the precision is not specified.
scale	INTEGER	Scale of the data type. 0 is used if the scale is not specified.
class_of	_db_class	Domain class if the data type is an object, **NULL** otherwise.
code_set	INTEGER	Character set (value of table "character sets supported by CUBRID" in _db_attribute) if it is character data type. 0 otherwise.
set_domains	SEQUENCE OF _db_domain	Domain information about the data type of collection element if it is collection data type. **NULL** otherwise.

_db_method

Represents information about the method. Indexes for class_of and meth_name are created.

Attribute Name	Data Type	Description
class_of	_db_class	Class to which the method belongs
meth_type	INTEGER	Type of the method defined in the class. 0 for an instance method, and 1 for a class method.
from_class_of	_db_class	If the method is inherited, the super class in which it is defined is used otherwise **NULL**
from_meth_name	VARCHAR(255)	If the method is inherited and its name is changed to resolve a name conflict, the original name defined in the super class is used otherwise **NULL**
meth_name	VARCHAR(255)	Method name
signatures	SEQUENCE OF _db_meth_sig	C function executed when the method is called

Example

The following is an example of retrieving class methods of the class with a class method (c.class_meth_count > 0), among classes owned by user 'DBA.'

```
SELECT class_name, SEQUENCE(SELECT meth_name
                            FROM _db_method m
                            WHERE m in c.class_meths)
FROM _db_class c
WHERE c.owner.name = 'DBA' AND c.class_meth_count > 0
ORDER BY 1;
  class_name            sequence((select meth_name from _db_method m where m in
c.class_meths))
============================================
  'db_serial'           {'change_serial_owner'}
  'db_authorizations'   {'add_user', 'drop_user', 'find_user', 'print_authorizations',
'info', 'change_owner', 'change_trigg
r_owner', 'get_owner'}
  'db_authorization'    {'check_authorization'}
  'db_user'             {'add_user', 'drop_user', 'find_user', 'login'}
  'db_root'             {'add_user', 'drop_user', 'find_user', 'print_authorizations',
'info', 'change_owner', 'change_trigg
r_owner', 'get_owner', 'change_sp_owner'}
```

_db_meth_sig

Represents information about the C function of the method. An index for meth_of is created.

Attribute Name	Data Type	Description
meth_of	_db_method	Method for the function information
arg_count	INTEGER	The number of input arguments of the function
func_name	VARCHAR(255)	Function name
return_value	SEQUENCE OF _db_meth_arg	Return value of the function
arguments	SEQUENCE OF _db_meth_arg	Input arguments of the function

_db_meth_arg

Represents information about the method argument. An index for meth_sig_of is created.

Attribute Name	Data Type	Description
meth_sig_of	_db_meth_sig	Information of the function to which the argument belongs
data_type	INTEGER	Data type of the argument (a value in the "Value" column of the "Data Types Supported by CUBRID" in _db_attribute)
index_of	INTEGER	Order of the argument listed in the function definition. Begins with 0 if it is a return value, and 1 if it is an input argument.
domains	SEQUENCE OF _db_domain	Domain of the argument

_db_meth_file

Represents information about the file in which the function is defined. An index for class_of is created.

Attribute Name	Data Type	Description
class_of	_db_class	Class to which the method file information belongs
from_class_of	_db_class	If the file information is inherited, the super class in which it is defined is used otherwise, **NULL**
path_name	VARCHAR(255)	File path in which the method is located

_db_query_spec

Represents the SQL definition statement of the virtual class. An index for class_of is created.

Attribute Name	Data Type	Description
class_of	_db_class	Class information of the virtual class
spec	VARCHAR(4096)	SQL definition statement of the virtual class

_db_index

Represents information about the index. An index for class_of is created.

Attribute Name	Data Type	Description
class_of	_db_class	Class to which to index belongs
index_name	varchar(255)	Index name
is_unique	INTEGER	1 if the index is unique, and 0 otherwise.
key_count	INTEGER	The number of attributes that comprise the key
key_attrs	SEQUENCE OF _db_index_key	Attributes that comprise the key
is_reverse	INTEGER	1 for a reverse index, and 0 otherwise.
is_primary_key	INTEGER	1 for a primary key, and 0 otherwise.
is_foreign_key	INTEGER	1 for a foreign key, and 0 otherwise.

Example

The following is an example of retrieving names of indexes that belong to the class.

```
SELECT class_of.class_name, index_name
FROM _db_index
ORDER BY 1;
  class_of.class_name   index_name
============================================
  '_db_attribute'       'i__db_attribute_class_of_attr_name'
  '_db_auth'            'i__db_auth_grantee'
  '_db_class'           'i__db_class_class_name'
  '_db_domain'          'i__db_domain_object_of'
  '_db_index'           'i__db_index_class_of'
  '_db_index_key'       'i__db_index_key_index_of'
  '_db_meth_arg'        'i__db_meth_arg_meth_sig_of'
  '_db_meth_file'       'i__db_meth_file_class_of'
  '_db_meth_sig'        'i__db_meth_sig_meth_of'
  '_db_method'          'i__db_method_class_of_meth_name'
  '_db_partition'       'i__db_partition_class_of_pname'
  '_db_query_spec'      'i__db_query_spec_class_of'
  '_db_stored_procedure'  'u__db_stored_procedure_sp_name'
  '_db_stored_procedure_args'  'i__db_stored_procedure_args_sp_name'
  'athlete'             'pk_athlete_code'
  'db_serial'           'pk_db_serial_name'
  'db_user'             'i_db_user_name'
  'event'               'pk_event_code'
  'game'                'pk_game_host_year_event_code_athlete_code'
  'game'                'fk_game_event_code'
  'game'                'fk_game_athlete_code'
  'history'             'pk_history_event_code_athlete'
  'nation'              'pk_nation_code'
  'olympic'             'pk_olympic_host_year'
  'participant'         'pk_participant_host_year_nation_code'
  'participant'         'fk_participant_host_year'
  'participant'         'fk_participant_nation_code'
  'record'              'pk_record_host_year_event_code_athlete_code_medal'
  'stadium'             'pk_stadium_code'
```

_db_index_key

Represents key information of the index. An index for index_of is created.

Attribute Name	Data Type	Description
index_of	_db_index	Index to which the key attribute belongs
key_attr_name	VARCHAR(255)	Name of the attribute that comprises the key
key_order	INTEGER	Order of the attribute in the key. Begins with 0.
asc_desc	INTEGER	1 if the order of attribute values is descending, and 0 otherwise.
key_prefix_length	INTEGER	Length of prefix to be used as a key

Example

The following is an example of retrieving names of indexes that belong to the class.

```
SELECT class_of.class_name, SEQUENCE(SELECT key_attr_name
                                     FROM _db_index_key k
                                     WHERE k in i.key_attrs)
FROM _db_index i
WHERE key_count >= 2;
  class_of.class_name   sequence((select key_attr_name from _db_index_key k where k in
i.key_attrs))
============================================
  '_db_partition'       {'class_of', 'pname'}
  '_db_method'          {'class_of', 'meth_name'}
  '_db_attribute'       {'class_of', 'attr_name'}
  'participant'         {'host_year', 'nation_code'}
  'game'                {'host_year', 'event_code', 'athlete_code'}
  'record'              {'host_year', 'event_code', 'athlete_code', 'medal'}
  'history'             {'event_code', 'athlete'}
```

_db_auth

Represents user authorization information of the class. An index for the grantee is created.

Attribute Name	Data Type	Description
grantor	db_user	Authorization grantor
grantee	db_user	Authorization grantee
class_of	_db_class	Class object to which authorization is to be granted
auth_type	VARCHAR(7)	Type name of the authorization granted
is_grantable	INTEGER	1 if authorization for the class can be granted to other users, and 0 otherwise.

Authorization types supported by CUBRID are as follows:

- **SELECT**
- **INSERT**
- **UPDATE**
- **DELETE**
- **ALTER**
- **INDEX**
- **EXECUTE**

Example

The following is an example of retrieving the authorization information defined in the class 'db_trig'.

```
SELECT grantor.name, grantee.name, auth_type
FROM _db_auth
WHERE class_of.class_name = 'db_trig';
  grantor.name          grantee.name          auth_type
==================================================================
  'DBA'                 'PUBLIC'              'SELECT'
```

_db_data_type

Represents the data type supported by CUBRID (see the "Data Types Supported by CUBRID" table in _db_attribute).

Attribute Name	Data Type	Description
type_id	INTEGER	Data type identifier. Corresponds to the "Value" column in the "Data Types Supported by CUBRID" table.
type_name	VARCHAR(9)	Data type name. Corresponds to the "Meaning" column in the "Data Types Supported by CUBRID" table.

Example

The following is an example of retrieving attributes and type names of the 'event' class.

```
SELECT a.attr_name, t.type_name
FROM _db_attribute a join _db_data_type t ON a.data_type = t.type_id
WHERE class_of.class_name = 'event'
ORDER BY a.def_order;
  attr_name             type_name
============================================
  'code'                'INTEGER'
  'sports'              'STRING'
  'name'                'STRING'
  'gender'              'CHAR'
  'players'             'INTEGER'
```

_db_partition

Represents information about partitions. Indexes for class_of and pname are created.

Attribute Name	Data Type	Description
class_of	_db_class	OID of the parent class
pname	VARCHAR(255)	Parent - **NULL**
ptype	INTEGER	0 - HASH 1 - RANGE 2 - LIST
pexpr	VARCHAR(255)	Parent only
pvalues	SEQUENCE OF	Parent - Column name, Hash size RANGE - MIN/MAX value : - Infinite MIN/MAX is saved as **NULL** LIST - value list

_db_stored_procedure

Represents information about Java stored procedures. An index for sp_name is created.

Attribute Name	Data Type	Description
sp_name	VARCHAR(255)	Stored procedure name
sp_type	INTEGER	Stored procedure type (function or procedure)
return_type	INTEGER	Return value type
arg_count	INTEGER	The number of arguments
args	SEQUENCE OF _db_stored_procedure_args	Argument list
lang	INTEGER	Implementation language (currently, Java)

target	VARCHAR(4096)	Name of the Java method to be executed
owner	db_user	Owner

_db_stored_procedure_args

Represents information about the Java stored procedure arguments. An index for sp_name is created.

Attribute Name	Data Type	Description
sp_name	VARCHAR(255)	Stored procedure name
index_of	INTEGER	Order of the arguments
arg_name	VARCHAR(255)	Argument name
data_type	INTEGER	Data type of the argument
mode	INTEGER	Mode (IN, OUT, INOUT)

db_user

Attribute Name	Data Type	Description
name	VARCHAR(1073741823)	User name
id	INTEGER	User identifier
password	db_password	User password. Not displayed to the user.
direct_groups	SET OF db_user	Groups to which the user belongs directly
groups	SET OF db_user	Groups to which the user belongs directly or indirectly
authorization	db_authorization	Information of the authorization owned by the user
triggers	SEQUENCE OF object	Triggers that occur due to user actions

Function Name

- **set_password**()
- **set_password_encoded**()
- **add_member**()
- **drop_member**()
- **print_authorizations**()
- **add_user**()
- **drop_user**()
- **find_user**()
- **login**()

db_authorization

Attribute Name	Data Type	Description
owner	db_user	User information
grants	SEQUENCE OF object	Sequence of {object for which the user has authorization, authorization grantor of the object, authorization type}

Method Name

- **check_authorization**(varchar(255), integer)

db_trigger

Attribute Name	Data Type	Description
owner	db_user	Trigger owner
name	VARCHAR(1073741823)	Trigger name
status	INTEGER	1 for INACTIVE, and 2 for ACTIVE. The default value is 2.
priority	DOUBLE	Execution priority between triggers. The default value is 0.
event	INTEGER	0 is set for UPDATE, 1 for UPDATE STATEMENT, 2 for DELETE, 3 for DELETE STATEMENT, 4 for INSERT, 5 for INSERT STATEMENT, 8 for COMMIT, and 9 for ROLLBACK.
target_class	object	Class object for the trigger target class
target_attribute	VARCHAR(1073741823)	Trigger target attribute name. If the target attribute is not specified, **NULL** is used.
target_class_attribute	INTEGER	If the target attribute is an instance attribute, 0 is used. If it is a class attribute, 1 is used. The default value is 0.
condition_type	INTEGER	If a condition exist, 1; otherwise **NULL**.
condition	VARCHAR(1073741823)	Action condition specified in the IF statement
condition_time	INTEGER	1 for BEFORE, 2 for AFTER, and 3 for DEFERRED if a condition exists; **NULL**, otherwise.
action_type	INTEGER	1 for one of INSERT, UPDATE, DELETE, CALL and EVALUATE, 2 for REJECT, 3 for INVALIDATE_TRANSACTION, and 4 for PRINT.
action_definition	VARCHAR(1073741823)	Execution statement to be triggered
action_time	INTEGER	1 for BEFORE, 2 for AFTER, and 3 for DEFERRED.

db_ha_apply_info

A table that saves the progress status every time the **applylogdb** utility applies replication logs. This table is updated at every point the **applylogdb** utility commits, and the acculmative count of operations are stored in the *_counter column. The meaning of each column is as follows:

Column Name	Column Type	Meaning
db_name	VARCHAR(255)	Name of the database saved in the log
db_creation_time	DATETIME	Creation time of the source database for the log to be applied
copied_log_path	VARCHAR(4096)	Path to the log file to be applied
page_id	INTEGER	Page of the replication log committed in the slave database
offset	INTEGER	Offset of the replication log committed in the slave database
log_record_time	DATETIME	Timestamp included in replication log committed in the slave database, i.e. the creation time of the log
last_access_time	DATETIME	Time when applylogdb was committed in the slave database
insert_counter	BIGINT	Number of times that applylogdb was inserted
update_counter	BIGINT	Number of times that applylogdb was updated
delete_counter	BIGINT	Number of times that applylogdb was deleted

schema_counter	BIGINT	Number of times that applylogdb changed the schema
commit_counter	BIGINT	Number of times that applylogdb was committed
fail_counter	BIGINT	Number of times that applylogdb failed to be inserted/updated/deleted/committed and to change the schema
required_page_id	INTEGER	Minimum pageid that applylogdb can read
start_time	DATETIME	Time when the applylogdb process accessed the slave database
status	INTEGER	Progress status (0: IDLE, 1: BUSY)

System Catalog Virtual Class

System Catalog Virtual Class

General users can only see information of classes for which they have authorization through system catalog virtual classes.

This section explains which information each system catalog virtual class represents, and virtual class definition statements.

DB_CLASS

Represents information of the classes for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
class_name	VARCHAR(255)	Class name
owner_name	VARCHAR(255)	Name of class owner
class_type	VARCHAR(6)	'CLASS' for a class, and 'VCLASS' for a virtual class
is_system_class	VARCHAR(3)	'YES' for a system class, and ‘NO’ otherwise.
partitioned	VARCHAR(3)	'YES' for a partitioned group class, and 'NO' otherwise.
is_reuse_oid_class	VARCHAR(3)	'YES' for a REUSE_OID class, and 'NO' otherwise.

Definition

```
CREATE VCLASS db_class (class_name, owner_name, class_type, is_system_class, partitioned,
is_reuse_oid_class)
AS

SELECT c.class_name, CAST(c.owner.name AS VARCHAR(255)),
    CASE c.class_type WHEN 0 THEN 'CLASS' WHEN 1 THEN 'VCLASS' ELSE 'UNKNOW' END,
    CASE WHEN MOD(c.is_system_class, 2) = 1 THEN 'YES' ELSE 'NO' END,
    CASE WHEN c.sub_classes IS NULL THEN 'NO' ELSE NVL((SELECT 'YES' FROM _db_partition p
WHERE p.class_of = c and p.pname IS NULL), 'NO') END,
    CASE WHEN MOD(c.is_system_class / 8, 2) = 1 THEN 'YES' ELSE 'NO' END
FROM _db_class c
WHERE CURRENT_USER = 'DBA' OR
    {c.owner.name} SUBSETEQ (
        SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})
        FROM db_user u, TABLE(groups) AS t(g)
        WHERE u.name = CURRENT_USER) OR
    {c} SUBSETEQ (
        SELECT SUM(SET{au.class_of})
        FROM _db_auth au
        WHERE {au.grantee.name} SUBSETEQ(
            SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})
            FROM db_user u, TABLE(groups) AS t(g)
            WHERE u.name = CURRENT_USER) AND  au.auth_type = 'SELECT');
```

Example 1

The following is an example of retrieving classes owned by the current user.

```
SELECT class_name
FROM db_class
WHERE owner_name = CURRENT_USER;
  class_name
======================
  'stadium'
  'code'
  'nation'
  'event'
  'athlete'
  'participant'
  'olympic'
  'game'
  'record'
  'history'
'female_event'
```

Note All examples of system catalog classes have been written in the csql utility. In this example, the user option is omitted (if omitted, the default user is **PUBLIC**). If not otherwise specified, **--no-auto-commit** (No auto-commit mode) and **-u** (Specify the user **dba**) options are used.

% csql --no-auto-commit -u dba demo

Example 2

The following is an example of retrieving virtual classes that can be accessed by the current user.

```
SELECT class_name
FROM db_class
WHERE class_type = 'VCLASS';
  class_name
======================
  'db_stored_procedure_args'
  'db_stored_procedure'
  'db_partition'
  'db_trig'
  'db_auth'
  'db_index_key'
  'db_index'
  'db_meth_file'
  'db_meth_arg_setdomain_elm'
  'db_meth_arg'
  'db_method'
  'db_attr_setdomain_elm'
  'db_attribute'
  'db_vclass'
  'db_direct_super_class'
  'db_class'
```

The following is an example of retrieving system classes that can be accessed by the current user user (**PUBLIC** user).

```
SELECT class_name
FROM db_class
WHERE is_system_class = 'YES' AND class_type = 'CLASS'
ORDER BY 1;
  class_name
======================
  'db_authorization'
  'db_authorizations'
  'db_root'
  'db_serial'
  'db_user'
```

DB_DIRECT_SUPER_CLASS

Represents names of super classes (if any) of the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
class_name	VARCHAR(255)	Class name
super_class_name	VARCHAR(255)	super class name

Definition

```
CREATE VCLASS db_direct_super_class (class_name, super_class_name)
AS
SELECT c.class_name, s.class_name
FROM _db_class c, TABLE(c.super_classes) AS t(s)
WHERE (CURRENT_USER = 'DBA' OR
              {c.owner.name} subseteq (
                              SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}),
set{})
                              from db_user u, table(groups) as t(g)
                              where u.name = CURRENT_USER ) OR
              {c} subseteq (
SELECT sum(set{au.class_of})
                              FROM _db_auth au
                              WHERE {au.grantee.name} subseteq (
                                                        SELECT set{CURRENT_USER} +
coalesce(sum(set{t.g.name}), set{})
                                                         from db_user u, table(groups) as t(g)
                                                        where u.name = CURRENT_USER ) AND
                                                                         au.auth_type =
'SELECT'))
```

Example

The following is an example of retrieving super classes of the 'female_event' class (see ADD SUPERCLASS Clause).

```
SELECT super_class_name
FROM db_direct_super_class
WHERE class_name = 'female_event';
  super_class_name
======================
  'event'
```

The following is an example of retrieving super classes of the class owned by the current user (**PUBLIC** user).

```
SELECT c.class_name, s.super_class_name
FROM db_class c, db_direct_super_class s
WHERE c.class_name = s.class_name AND c.owner_name = user
ORDER BY 1;
  class_name            super_class_name
============================================
  'female_event'        'event'
```

DB_VCLASS

Represents SQL definition statements of virtual classes for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
vclass_name	VARCHAR(255)	Virtual class name
vclass_def	VARCHAR(4096)	SQL definition statement of the virtual class

Definition

```
CREATE VCLASS db_vclass (vclass_name, vclass_def)
AS
SELECT q.class_of.class_name, q.spec
FROM _db_query_spec q
WHERE CURRENT_USER = 'DBA' OR
        {q.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
```

```
                where u.name = CURRENT_USER ) OR
        {q.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT');
```

Example

The following is an example of retrieving SQL definition statements of the 'db_class' virtual class.

```
SELECT vclass_def
FROM db_vclass
WHERE vclass_name = 'db_class';
'SELECT c.class_name, CAST(c.owner.name AS VARCHAR(255)), CASE c.class_type WHEN 0 THEN
'CLASS' WHEN 1 THEN 'VCLASS' WHEN 2 THEN 'PROXY' ELSE 'UNKNOW' END, CASE WHEN
MOD(c.is_system_class, 2) = 1 THEN 'YES' ELSE 'NO' END, CASE WHEN c.sub_classes IS NULL
THEN 'NO' ELSE NVL((SELECT 'YES' FROM _db_partition p WHERE p.class_of = c and p.pname IS
NULL), 'NO') END FROM _db_class c WHERE CURRENT_USER = 'DBA' OR {c.owner.name} SUBSETEQ
(  SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})  FROM db_user u,
TABLE(groups) AS t(g)  WHERE u.name = CURRENT_USER) OR {c} SUBSETEQ (  SELECT
SUM(SET{au.class_of})  FROM _db_auth au  WHERE {au.grantee.name} SUBSETEQ (  SELECT
SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})  FROM db_user u, TABLE(groups) AS
t(g)  WHERE u.name = CURRENT_USER) AND  au.auth_type = 'SELECT')'
```

DB_ATTRIBUTE

Represents the attribute information of the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
attr_name	VARCHAR(255)	Attribute name
class_name	VARCHAR(255)	Name of the class to which the attribute belongs
attr_type	VARCHAR(8)	'INSTANCE' for an instance attribute, 'CLASS' for a class attribute, and 'SHARED' for a shared attribute.
def_order	INTEGER	Order of attributes in the class. Begins with 0. If the attribute is inherited, the order is the one defined in the super class.
from_class_name	VARCHAR(255)	If the attribute is inherited, the super class in which it is defined is used. Otherwise, **NULL**
from_attr_name	VARCHAR(255)	If the attribute is inherited and its name is changed to resolve a name conflict, the original name defined in the super class is used. Otherwise, **NULL**
data_type	VARCHAR(9)	Data type of the attribute (one in the "Meaning" column of the "Data Types Supported by CUBRID" table in _db_attribute)
prec	INTEGER	Precision of the data type. 0 is used if the precision is not specified.
scale	INTEGER	Scale of the data type. 0 is used if the scale is not specified.
code_set	INTEGER	Character set (value of table "character sets supported by CUBRID" in _db_attribute) if it is string type. 0 otherwise.
domain_class_name	VARCHAR(255)	Domain class name if the data type is an object. **NULL** otherwise.
default_value	VARCHAR(255)	Saved as a character string by default, regardless of data types. If no default value is specified, **NULL** is saved if a default value is **NULL**, it is displayed as 'NULL'. An object data type is represented as 'volume id | page id | slot id' while a set data type is represented as '{element 1, element 2, ... '.
is_nullable	VARCHAR(3)	'NO' if a not null constraint is set, and 'YES' otherwise.

Definition

```
CREATE VCLASS db_attribute (
attr_name, class_name, attr_type, def_order, from_class_name, from_attr_name, data_type,
prec, scale, code_set, domain_class_name, default_value, is_nullable)
AS
SELECT a.attr_name, c.class_name,
            CASE WHEN a.attr_type = 0 THEN 'INSTANCE'
                      WHEN a.attr_type = 1 THEN 'CLASS'
                      ELSE 'SHARED' END,
            a.def_order, a.from_class_of.class_name, a.from_attr_name, t.type_name,
            d.prec, d.scale, d.code_set, d.class_of.class_name, a.default_value,
            CASE WHEN a.is_nullable = 0 THEN 'YES' ELSE 'NO' END
FROM _db_class c, _db_attribute a, _db_domain d, _db_data_type t
WHERE a.class_of = c AND d.object_of = a AND d.data_type = t.type_id AND
              (CURRENT_USER = 'DBA' OR
              {c.owner.name} subseteq (
                              SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}),
set{})
                              from db_user u, table(groups) as t(g)
                              where u.name = CURRENT_USER ) OR
              {c} subseteq (
SELECT sum(set{au.class_of})
                              FROM _db_auth au
                              WHERE {au.grantee.name} subseteq (
                                                      SELECT set{CURRENT_USER} +
coalesce(sum(set{t.g.name}), set{})
                                                       from db_user u, table(groups) as t(g)
                                                      where u.name = CURRENT_USER ) AND
                                                                      au.auth_type =
'SELECT'))
```

Example 1

The following is an example of retrieving attributes and data types of the ‘event’ class.

```
SELECT attr_name, data_type, domain_class_name
FROM db_attribute
WHERE class_name = 'event'
ORDER BY def_order;
  attr_name             data_type             domain_class_name
==================================================================
  'code'                'INTEGER'             NULL
  'sports'              'STRING'              NULL
  'name'                'STRING'              NULL
  'gender'              'CHAR'                NULL
  'players'             'INTEGER'             NULL
```

Example 2

The following is an example of retrieving attributes of the ‘female_event’ class and its super class.

```
SELECT attr_name, from_class_name
FROM db_attribute
WHERE class_name = 'female_event'
ORDER BY def_order;
  attr_name             from_class_name
============================================
  'code'                'event'
  'sports'              'event'
  'name'                'event'
  'gender'              'event'
  'players'             'event'
  'players'             'event'
```

Example 3

The following is an example of retrieving classes whose attribute names are similar to 'name,' among the ones owned by the current user. (The user is **PUBLIC**.)

```
SELECT a.class_name, a.attr_name
FROM db_class c join db_attribute a ON c.class_name = a.class_name
```

```
WHERE c.owner_name = CURRENT_USER AND attr_name like '%name%'
ORDER BY 1;
  class_name            attr_name
============================================
  'athlete'             'name'
  'code'                'f_name'
  'code'                's_name'
  'event'               'name'
  'female_event'        'name'
  'nation'              'name'
  'stadium'             'name'
```

DB_ATTR_SETDOMAIN_ELM

Among attributes of the class to which the current user has access authorization in the database, if an attribute's data type is a set (set, multiset, sequence), this macro represents the data type of the element of the set.

Attribute Name	Data Type	Description
attr_name	VARCHAR(255)	Attribute name
class_name	VARCHAR(255)	Name of the class to which the attribute belongs
attr_type	VARCHAR(8)	'INSTANCE' for an instance attribute, 'CLASS' for a class attribute, and 'SHARED' for a shared attribute.
data_type	VARCHAR(9)	Data type of the element
prec	INTEGER	Precision of the data type of the element
scale	INTEGER	Scale of the data type of the element
code_set	INTEGER	Character set if the data type of the element is a character
domain_class_name	VARCHAR(255)	Domain class name if the data type of the element is an object

Definition

```
CREATE VCLASS db_attr_setdomain_elm (
attr_name, class_name, attr_type,data_type, prec, scale, code_set, domain_class_name)
AS
SELECT a.attr_name, c.class_name,
       CASE WHEN a.attr_type = 0 THEN 'INSTANCE'
            WHEN a.attr_type = 1 THEN 'CLASS'
            ELSE 'SHARED' END,
       et.type_name, e.prec, e.scale, e.code_set, e.class_of.class_name
FROM _db_class c, _db_attribute a, _db_domain d,
      TABLE(d.set_domains) AS t(e), _db_data_type et
WHERE a.class_of = c AND d.object_of = a AND e.data_type = et.type_id AND
        (CURRENT_USER = 'DBA' OR
        {c.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {c} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                        from db_user u, table(groups) as t(g)
                        where u.name = CURRENT_USER ) AND
                                au.auth_type = 'SELECT'));
```

If the set_attr attribute of class D is of a SET (A, B, C) type, the following three records exist.

Attr_name	Class_name	Attr_type	Data_type	prec	Scale	Code_set	Domain_class_name
'set_attr'	'D'	'INSTANCE'	'SET'	0	0	0	'A'
'set_attr'	'D'	'INSTANCE'	'SET'	0	0	0	'B'
'set_attr'	'D'	'INSTANCE'	'SET'	0	0	0	'C'

Example

The following is an example of retrieving set type attributes and data types of the 'city' class. (The city table defined in Containment Operators is created.)

```
SELECT attr_name, attr_type, data_type, domain_class_name
FROM db_attr_setdomain_elm
WHERE class_name = 'city';
  attr_name             attr_type             data_type             domain_class_name
==============================================================================

'sports'              'INSTANCE'            'STRING'              NULL
```

DB_METHOD

Represents the method information of the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
meth_name	VARCHAR(255)	Method name
class_name	VARCHAR(255)	Name of the class to which the method belongs
meth_type	VARCHAR(8)	'INSTANCE' for an instance method, and 'CLASS' for a class method.
from_class_name	VARCHAR(255)	If the method is inherited, the super class in which it is defined is used otherwise **NULL**
from_meth_name	VARCHAR(255)	If the method is inherited and its name is changed to resolve a name conflict, the original name defined in the super class is used otherwise **NULL**
func_name	VARCHAR(255)	Name of the C function for the method

Definition

```
CREATE VCLASS db_method (
meth_name, class_name, meth_type, from_class_name, from_meth_name, func_name)
AS

SELECT m.meth_name, m.class_of.class_name,
            CASE WHEN m.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
            m.from_class_of.class_name, m.from_meth_name, s.func_name
FROM _db_method m, _db_meth_sig s
WHERE s.meth_of = m AND
              (CURRENT_USER = 'DBA' OR
              {m.class_of.owner.name} subseteq (
                              SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}),
set{})
                              from db_user u, table(groups) as t(g)
                              where u.name = CURRENT_USER ) OR
              {m.class_of} subseteq (
SELECT sum(set{au.class_of})
                              FROM _db_auth au
                              WHERE {au.grantee.name} subseteq (
                                                        SELECT set{CURRENT_USER} +
coalesce(sum(set{t.g.name}), set{})
                                                         from db_user u, table(groups) as t(g)
                                                        where u.name = CURRENT_USER ) AND
                                                                        au.auth_type =
'SELECT'))
```

Example

The following is an example of retrieving methods of the 'db_user' class.

```
SELECT meth_name, meth_type, func_name
FROM db_method
WHERE class_name = 'db_user'
ORDER BY meth_type, meth_name;
  meth_name             meth_type             func_name
```

```
==================================================================
  'add_user'           'CLASS'              'au_add_user_method'
  'drop_user'          'CLASS'              'au_drop_user_method'
  'find_user'          'CLASS'              'au_find_user_method'
  'login'              'CLASS'              'au_login_method'
  'add_member'         'INSTANCE'           'au_add_member_method'
  'drop_member'        'INSTANCE'           'au_drop_member_method'
  'print_authorizations'  'INSTANCE'          'au_describe_user_method'
  'set_password'       'INSTANCE'           'au_set_password_method'
  'set_password_encoded'  'INSTANCE'          'au_set_password_encoded_method'
  'set_password_encoded_sha1'  'INSTANCE'          'au_set_password_encoded_sha1_method'
```

DB_METH_ARG

Represents the input/output argument information of the method of the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
meth_name	VARCHAR(255)	Method name
class_name	VARCHAR(255)	Name of the class to which the method belongs
meth_type	VARCHAR(8)	'INSTANCE' for an instance method, and 'CLASS' for a class method.
index_of	INTEGER	Order in which arguments are listed in the function definition. Begins with 0 if it is a return value, and 1 if it is an input argument.
data_type	VARCHAR(9)	Data type of the argument
prec	INTEGER	Precision of the argument
scale	INTEGER	Scale of the argument
code_set	INTEGER	Character set if the data type of the argument is a character.
domain_class_name	VARCHAR(255)	Domain class name if the data type of the argument is an object.

Definition

```
CREATE VCLASS db_meth_arg (
meth_name, class_name, meth_type,
index_of, data_type, prec, scale, code_set, domain_class_name)
AS
SELECT s.meth_of.meth_name, s.meth_of.class_of.class_name,
       CASE WHEN s.meth_of.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
       a.index_of, t.type_name, d.prec, d.scale, d.code_set,
       d.class_of.class_name
FROM _db_meth_sig s, _db_meth_arg a, _db_domain d, _db_data_type t
WHERE a.meth_sig_of = s AND d.object_of = a AND d.data_type = t.type_id AND
        (CURRENT_USER = 'DBA' OR
        {s.meth_of.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {s.meth_of.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

Example

The following is an example of retrieving input arguments of the method of the 'db_user' class.

```
SELECT meth_name, data_type, prec
FROM db_meth_arg
```

```
WHERE class_name = 'db_user';
  meth_name             data_type                    prec
=========================================================
  'append_data'         'STRING'               1073741823
```

DB_METH_ARG_SETDOMAIN_ELM

If the data type of the input/output argument of the method of the class is a set, for which the current user has access authorization in the database, this macro represents the data type of the element of the set.

Attribute Name	Data Type	Description
meth_name	VARCHAR(255)	Method name
class_name	VARCHAR(255)	Name of the class to which the method belongs
meth_type	VARCHAR(8)	‘INSTANCE’ for an instance method, and 'CLASS' for a class method.
index_of	INTEGER	Order of arguments listed in the function definition. Begins with 0 if it is a return value, and 1 if it is an input argument.
data_type	VARCHAR(9)	Data type of the element
prec	INTEGER	Precision of the element
scale	INTEGER	Scale of the element
code_set	INTEGER	Character set if the data type of the element is a character
domain_class_name	VARCHAR(255)	Domain class name if the data type of the element is an object

Definition

```
CREATE VCLASS db_meth_arg_setdomain_elm(
meth_name, class_name, meth_type,
index_of, data_type, prec, scale, code_set, domain_class_name)
AS
SELECT s.meth_of.meth_name, s.meth_of.class_of.class_name,
       CASE WHEN s.meth_of.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
       a.index_of, et.type_name, e.prec, e.scale, e.code_set,
       e.class_of.class_name
FROM _db_meth_sig s, _db_meth_arg a, _db_domain d,
      TABLE(d.set_domains) AS t(e), _db_data_type et
WHERE a.meth_sig_of = s AND d.object_of = a AND e.data_type = et.type_id AND
        (CURRENT_USER = 'DBA' OR
        {s.meth_of.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {s.meth_of.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

DB_METH_FILE

Represents information of the file where the method of the class for which the current user has access authorization in the database is defined.

Attribute Name	Data Type	Description
class_name	VARCHAR(255)	Name of the class to which the method file belongs
path_name	VARCHAR(255)	File path in which the C function is defined
from_class_name	VARCHAR(255)	Name of the super class in which the method file is defined if the

	method is inherited, and otherwise **NULL**

Definition

```
CREATE VCLASS db_meth_file (class_name, path_name, from_class_name)
AS
SELECT f.class_of.class_name, f.path_name, f.from_class_of.class_name
FROM _db_meth_file f
WHERE (CURRENT_USER = 'DBA' OR
              {f.class_of.owner.name} subseteq (
                              SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}),
set{})
                              from db_user u, table(groups) as t(g)
                              where u.name = CURRENT_USER ) OR
              {f.class_of} subseteq (
SELECT sum(set{au.class_of})
                              FROM _db_auth au
                              WHERE {au.grantee.name} subseteq (
                                                      SELECT set{CURRENT_USER} +
coalesce(sum(set{t.g.name}), set{})
                                                       from db_user u, table(groups) as t(g)
                                                      where u.name = CURRENT_USER ) AND
                                                              au.auth_type =
'SELECT'))
```

DB_INDEX

Represents information of indexes created for the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
index_name	VARCHAR(255)	Index name
is_unique	VARCHAR(3)	‘YES’ for a unique index, and ‘NO’ otherwise.
is_reverse	VARCHAR(3)	'YES’ for a reversed index, and ‘NO’ otherwise.
class_name	VARCHAR(255)	Name of the class to which the index belongs
key_count	INTEGER	The number of attributes that comprise the key
is_primary_key	VARCHAR(3)	'YES' for a primary key, and ‘NO’ otherwise.
is_foreign_key	VARCHAR(3)	'YES' for a foreign key, and ‘NO’ otherwise.

Definition

```
CREATE VCLASS db_index (index_name, is_unique, is_reverse, class_name, key_count,
is_primary_key, is_foreign_key)
AS
SELECT i.index_name, CASE WHEN i.is_unique = 0 THEN 'NO' ELSE 'YES' END,
CASE WHEN i.is_reverse = 0 THEN 'NO' ELSE 'YES' END, i.class_of.class_name, i.key_count,
CASE WHEN i.is_primary_key = 0 THEN 'NO' ELSE 'YES' END, CASE WHEN i.is_foreign_key = 0
THEN 'NO' ELSE 'YES' END
FROM _db_index i
WHERE (CURRENT_USER = 'DBA' OR
        {i.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {i.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                        from db_user u, table(groups) as t(g)
                        where u.name = CURRENT_USER ) AND
                                au.auth_type = 'SELECT'));
```

Example

The following is an example of retrieving index information of the class.

다음 예제에서는 클래스의 인덱스 정보를 검색한다.

```
SELECT class_name, index_name, is_unique
FROM db_index
ORDER BY 1;

  class_name            index_name            is_unique
==================================================================
  'athlete'             'pk_athlete_code'     'YES'
  'city'                'pk_city_city_name'   'YES'
  'db_serial'           'pk_db_serial_name'   'YES'
  'db_user'             'i_db_user_name'      'NO'
  'event'               'pk_event_code'       'YES'
  'female_event'        'pk_event_code'       'YES'
  'game'                'pk_game_host_year_event_code_athlete_code'  'YES'
  'game'                'fk_game_event_code'  'NO'
  'game'                'fk_game_athlete_code'  'NO'
  'history'             'pk_history_event_code_athlete'  'YES'
  'nation'              'pk_nation_code'      'YES'
  'olympic'             'pk_olympic_host_year'  'YES'
  'participant'         'pk_participant_host_year_nation_code'  'YES'
  'participant'         'fk_participant_host_year'  'NO'
  'participant'         'fk_participant_nation_code'  'NO'
  'record'              'pk_record_host_year_event_code_athlete_code_medal'  'YES'
  'stadium'             'pk_stadium_code'     'YES'
```

DB_INDEX_KEY

Represents the key information of indexes created for the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
index_name	VARCHAR(255)	Index name
class_name	VARCHAR(255)	Name of the class to which the index belongs
key_attr_name	VARCHAR(255)	Name of attributes that comprise the key
key_order	INTEGER	Order of attributes in the key. Begins with 0.
asc_desc	VARCHAR(4)	'DESC' if the order of attribute values is descending, and 'ASC' otherwise.
key_prefix_length	INTEGER	Length of prefix to be used as a key

Definition

```
CREATE VCLASS db_index_key (index_name, class_name, key_attr_name, key_order,
key_prefix_length)
AS
SELECT k.index_of.index_name, k.index_of.class_of.class_name, k.key_attr_name, k.key_order
CASE k.asc_desc
WHEN 0 THEN 'ASC'
WHEN 1 THEN 'DESC' ELSE 'UNKN' END,
k.key_prefix_length
FROM _db_index_key k
WHERE (CURRENT_USER = 'DBA' OR
    {k.index_of.class_of.owner.name}
    subseteq (
        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
        from db_user u, table(groups) as t(g)
        where u.name = CURRENT_USER ) OR {k.index_of.class_of}
        subseteq (
            SELECT sum(set{au.class_of})
            FROM _db_auth au
            WHERE {au.grantee.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
```

```
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) AND
            au.auth_type = 'SELECT'));
```

Example

The following is an example of retrieving index key information of the class.

```
SELECT class_name, key_attr_name, index_name
FROM db_index_key
ORDER BY class_name, key_order;
  'athlete'             'code'                'pk_athlete_code'
  'city'                'city_name'           'pk_city_city_name'
  'db_serial'           'name'                'pk_db_serial_name'
  'db_user'             'name'                'i_db_user_name'
  'event'               'code'                'pk_event_code'
  'female_event'        'code'                'pk_event_code'
  'game'                'host_year'           'pk_game_host_year_event_code_athlete_code'
  'game'                'event_code'          'fk_game_event_code'
  'game'                'athlete_code'        'fk_game_athlete_code'
...
```

DB_AUTH

Represents authorization information of the classes for which the current user has authorization in the database.

Attribute Name	Data Type	Description
grantor_name	VARCHAR(255)	Name of the user who grants authorization
grantee_name	VARCHAR(255)	Name of the user who is granted authorization
class_name	VARCHAR(255)	Name of the class for which authorization is to be granted
auth_type	VARCHAR(7)	Name of the authorization type granted
is_grantable	VARCHAR(3)	'YES' if authorization for the class can be granted to other users, and 'NO' otherwise.

Definition

```
CREATE VCLASS db_auth (grantor_name, grantee_name, class_name, auth_type, is_grantable )
AS
SELECT CAST(a.grantor.name AS VARCHAR(255)),
        CAST(a.grantee.name AS VARCHAR(255)),
        a.class_of.class_name, a.auth_type,
        CASE WHEN a.is_grantable = 0 THEN 'NO' ELSE 'YES' END
FROM _db_auth a
WHERE (CURRENT_USER = 'DBA' OR
        {a.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {a.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

Example

The following is an example of retrieving authorization information of the classes whose names begin with 'db_a'.

```
SELECT class_name, auth_type, grantor_name
FROM db_auth
WHERE class_name like 'db_a%'
ORDER BY 1;
  class_name            auth_type             grantor_name
==================================================================
```

```
  'db_attr_setdomain_elm'  'SELECT'             'DBA'
  'db_attribute'           'SELECT'             'DBA'
  'db_auth'                'SELECT'             'DBA'
  'db_authorization'       'EXECUTE'            'DBA'
  'db_authorization'       'SELECT'             'DBA'
  'db_authorizations'      'EXECUTE'            'DBA'
  'db_authorizations'      'SELECT'             'DBA'
```

DB_TRIG

Represents information of the trigger that has the class for which the current user has access authorization in the database, or its attribute as the target.

Attribute Name	Data Type	Description
trigger_name	VARCHAR(255)	Trigger name
target_class_name	VARCHAR(255)	Target class
target_attr_name	VARCHAR(255)	Target attribute. If not specified in the trigger, **NULL**
target_attr_type	VARCHAR(8)	Target attribute type. If specified, 'INSTANCE' is used for an instance attribute, and 'CLASS' is used for a class attribute.
action_type	INTEGER	1 for one of INSERT, UPDATE, DELETE, CALL and EVALUATE, 2 for REJECT, 3 for INVALIDATE_TRANSACTION, and 4 for PRINT.
action_time	INTEGER	1 for BEFORE, 2 for AFTER, and 3 for DEFERRED.

Example

- The following is an example of showing information of the trigger that has the class for which the current user has access authorization, or its attribute as the target.

```
CREATE VCLASS db_trig (
trigger_name, target_class_name, target_attr_name, target_attr_type, action_type,
action_time)
AS
SELECT CAST(t.name AS VARCHAR(255)), c.class_name,
        CAST(t.target_attribute AS VARCHAR(255)),
        CASE WHEN t.target_class_attribute = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
        t.action_type, t.action_time
FROM _db_class c, db_trigger t
WHERE t.target_class = c.class_of AND
        (CURRENT_USER = 'DBA' OR
        {c.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {c} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                        from db_user u, table(groups) as t(g)
                        where u.name = CURRENT_USER ) AND
                                au.auth_type = 'SELECT'));
```

DB_PARTITION

Represents information of partitioned classes for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
class_name	VARCHAR(255)	Class name
partition_name	VARCHAR(255)	Partition name
partition_class_name	VARCHAR(255)	Partitioned class name

partition_type	VARCHAR(32)	Partition type (HASH, RANGE, LIST)
partition_expr	VARCHAR(255)	Partition expression
partition_values	SEQUENCE OF	RANGE ? MIN/MAX value - For infinite MIN/MAX, **NULL** LIST - value list

Definition

```
CREATE VCLASS db_partition
(sp_name, sp_type, return_type, arg_count, lang, target, owner)
AS
SELECT p.class_of.class_name AS class_name, p.pname AS partition_name,
            p.class_of.class_name || '__p__' || p.pname AS partition_class_name,
            CASE WHEN p.ptype = 0 THEN 'HASH'
                 WHEN p.ptype = 1 THEN 'RANGE'
            ELSE 'LIST' ENDASpartition_type,
            TRIM(SUBSTRING( pi.pexpr FROM 8 FOR (POSITION(' FROM ' IN pi.pexpr)-8))) AS
               partition_expression,
            p.pvalues AS partition_values
FROM _db_partition p,
     ( select * from _db_partition sp
where sp.class_of =  p.class_of AND sp.pname is null) pi
WHERE p.pname is not null AND
      ( CURRENT_USER = 'DBA'
        OR
        {p.class_of.owner.name} SUBSETEQ
         ( SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})
           FROM db_user u, TABLE(groups) AS t(g)
           WHERE u.name = CURRENT_USER
         )
        OR
        {p.class_of} SUBSETEQ
         ( SELECT SUM(SET{au.class_of})
           FROM _db_auth au
           WHERE {au.grantee.name} SUBSETEQ
                 ( SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})
                   FROM db_user u, TABLE(groups) AS t(g)
                   WHERE u.name = CURRENT_USER) AND
                   au.auth_type = 'SELECT'
         )
      )
```

Example

The following is an example of retrieving the partition information currently configured for the participant2 class (see examples in Defining Range Partitions).

```
SELECT * FROM db_partition WHERE class_name = 'participant2';
  class_name            partition_name        partition_class_name         partition_type
  partition_expr        partition_values
==========================================================================================
==========================================
  'participant2'        'before_2000'         'participant2__p__before_2000'  'RANGE'
  'host_year'           {NULL, 2000}
  'participant2'        'before_2008'         'participant2__p__before_2008'  'RANGE'
  'host_year'           {2000, 2008}
```

DB_STORED_PROCEDURE

Represents information of Java stored procedures for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
sp_name	VARCHAR(255)	Stored procedure name
sp_type	VARCHAR(16)	Stored procedure type (function or procedure)

return_type	VARCHAR(16)	Return value type
arg_count	INTEGER	The number of arguments
lang	VARCHAR(16)	Implementing language (currently, Java)
target	VARCHAR(4096)	Name of the Java method to be executed
owner	VARCHAR(256)	Owner

Definition

```
CREATE VCLASS db_stored_procedure
(sp_name, sp_type, return_type, arg_count, lang, target, owner)
AS
SELECT sp.sp_name,
            CASE sp.sp_type   WHEN 1 THEN 'PROCEDURE'
            ELSE 'FUNCTION' END,
            CASE WHEN sp.return_type = 0 THEN 'void'
                 WHEN sp.return_type = 28 THEN 'CURSOR'
            ELSE ( SELECT dt.type_name
                   FROM _db_data_type dt
                   WHERE sp.return_type = dt.type_id) END,
           sp.arg_count,
           CASE sp.lang   WHEN 1 THEN 'JAVA'
           ELSE '' END, sp.target, sp.owner.name
FROM _db_stored_procedure sp
```

Example

The following is an example of retrieving Java stored procedures owned by the current user.

```
SELECT sp_name, target from db_stored_procedure WHERE sp_type = 'FUNCTION' AND owner =
CURRENT_USER
  sp_name               target
============================================
  'hello'               'SpCubrid.HelloCubrid() return java.lang.String'
  'sp_int'              'SpCubrid.SpInt(int) return int'
```

DB_STORED_PROCEDURE_ARGS

Represents the argument information of Java stored procedures for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
sp_name	VARCHAR(255)	Stored procedure name
index_of	INTEGER	Order of the arguments
arg_name	VARCHAR(256)	Argument name
data_type	VARCHAR(16)	Data type of the argument
mode	VARCHAR(6)	Mode (IN, OUT, INOUT)

Definition

```
CREATE VCLASS db_stored_procedure_args (sp_name, index_of, arg_name, data_type, mode)
AS
SELECT sp.sp_name, sp.index_of, sp.arg_name,
            CASE sp.data_type   WHEN 28 THEN 'CURSOR'
            ELSE ( SELECT dt.type_name FROM _db_data_type dt
                   WHERE sp.data_type = dt.type_id) END,
            CASE WHEN sp.mode = 1 THEN 'IN' WHEN sp.mode = 2 THEN 'OUT'
            ELSE 'INOUT' END
FROM _db_stored_procedure_args sp
ORDER BY sp.sp_name, sp.index_of ;
```

Example

The following is an example of retrieving arguments the 'phone_info' Java stored procedure in the order of the arguments.

```
SELECT index_of, arg_name, data_type, mode FROM db_stored_procedure_args
WHERE sp_name = 'phone_info'
ORDER BY index_of
     index_of  arg_name              data_type             mode
===============================================================
            0  'name'                'STRING'              'IN'
            1  'phoneno'             'STRING'              'IN'
```

Catalog Class/Virtual Class Authorization

Catalog classes are created to be owned by **DBA**. However, **DBA** can only execute **SELECT** operations. If **DBA** executes operations such as **UPDATE**/**DELETE**, an authorization failure error occurs. General users cannot execute queries on system catalog classes.

Although catalog virtual classes are created to be owned by **DBA**, all users can perform the **SELECT** statement on catalog virtual classes. Of course, **UPDATE**/**DELETE** operations on catalog virtual classes are not allowed.

Updating catalog classes/virtual classes is automatically performed by the system when users execute a DDL statement that creates/modifies/deletes a class/attribute/index/user/authorization.

Consistency of Catalog Information

Catalog information is represented by the instance of a catalog class/virtual class. If such information is accessed at the **READ UNCOMMITTED INSTANCES** (**TRAN_REP_CLASS_UNCOMMIT_INSTANCE** or **TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE**) isolation level, incorrect values (values being changed) can be read. Therefore, to get correct catalog information, you must use the **SELECT** query on the catalog class/virtual class at the **READ COMMITTED INSTANCES** isolation level or higher.

Querying on Catalog

To query on catalog classes, you must convert identifiers such as class, virtual class, attribute, trigger, method and index names to lowercases, and create them. Therefore, you must use lowercases when querying on catalog classes.

```
CREATE TABLE Foo(name varchar(255));
SELECT class_name, partitioned FROM db_class WHERE class_name = 'Foo';
There are no results.
SELECT class_name, partitioned FROM db_class WHERE class_name = 'foo';
  class_name   partitioned
============================
  'foo'        'NO'
```

Administrator's Guide

Administrator's Guide

The "Administrator's Guide" provides the database administrators (**DBA**) with details on how to operate the CUBRID system. The guide includes instructions on the following: database management tasks (creating and deleting databases, adding volume, etc.), migration tasks (moving database to a different location or making changes so that it fits the system's version), and making back-ups and rollbacks of the database in case of failures.

It also includes instructions on how to use the CUBRID utilities, which starts and stops various processes of the CUBRID server, the broker and manager server.

This chapter contains the following:

- How to use CUBRID utilities
- How to control the CUBRID (service, database server, broker, manager server)
- How to use the database administrative utilities
- Database migration
- Database backup and restore
- CUBRID HA

CUBRID Utilities

The CUBRID utilities provide features that can be used to comprehensively manage the CUBRID service. CUBRID utilities are divided into the service management utility, which is used to manage the CUBRID service process, and the database management utility, which is used to manage the database.

The service management utility is as follows:

- Service utility : Operates and manages the master process.
- cubrid service
- Server Utility : Operates and manages the server process.
- cubrid server
- Broker utility : Operates and manages the broker process and application server (CAS) process.
- cubrid broker
- Manager utility : Operates and manages the manager server process.
- cubrid manager
- HA utility : Operates an;d manages the HA related process.
- cubrid_heartbeat

See Registering Services for details.

The database management utility is as follows:

- Database create/add volume/delete utility
- cubrid createdb
- cubrid addvoldb
- cubrid deletedb
- Database rename/alter host/copy/install utility
- cubrid renamedb
- cubrid alterdbhost
- cubrid copydb
- cubrid installdb
- Database space check/space compaction utility
- cubrid spacedb
- cubrid compactdb
- Database query plan check/optimization utility
- cubrid plandump
- cubrid optimizedb
- cubrid statdump
- Database lock check/transaction kill/consistency check utility
- cubrid lockdb
- cubrid killtran
- cubrid checkdb
- Database diagnostics utility
- cubrid diagdb
- cubrid paramdump
- Database loading Utilities
- cubrid loaddb
- cubrid unloaddb
- Database backup/restore utility
- cubrid backupdb
- cubrid restoredb

- HA utilities
- cubrid changemode
- cubrid copylogdb
- cubrid applylogdb
- cubrid_applyinfo

See How to Use the CUBRID Management Utilities (Syntax) for details.

The following information will be displayed upon entering **cubrid** in the prompt.

```
% cubrid

cubrid utility, version R4.0
usage: cubrid <utility-name> [args]
Type 'cubrid <utility-name>' for help on a specific utility.

Available service's utilities:
    service
    server
    broker
    manager
    heartbeat

Available administrator's utilities:
    addvoldb
    alterdbhost
    backupdb
    checkdb
    compactdb
    copydb
    createdb
    deletedb
    diagdb
    installdb
    killtran
    loaddb
    lockdb
    optimizedb
    plandump
    renamedb
    restoredb
    spacedb
    unloaddb
    paramdump
    statdump
    changemode
    copylogdb
    applylogdb
    applyinfo

cubrid is a tool for DBMS.
For additional information, see http://www.cubrid.com
```

Remark

If you want to control the service using Cubrid utilities in Windows Vista and the later versions of Windows, you are recommended to open the command prompt window as an administrator.

If you do not use the command prompt window as an administrator to use Cubrid utilities, you can execute it through UAC (User Account Control) dialog, but cannot check the result message.

To open the command prompt window in Windows Vista and the later versions of Windows as an administrator, right-click [Start] > [All Programs] > [Accessories] > [Command Prompt] to select [Run as Administrator]. When the dialog box to check the elevation of rights pops up, click [Yes] to open the Command Prompt as an administrator.

CUBRID Controls

How to Use CUBRID Utilities (Syntax)

The following provides descriptions on how to use CUBRID utilities (syntaxes).

CUBRID Service Control

The following is the **cubrid** utility syntax used to control services registered in the CUBRID configuration file. The following can be used as *command*; **start** to start the service, **stop** to stop the service, **restart** to restart the service, **status** to verify the status. It is not required to enter additional options or arguments.

```
cubrid service command
command : { start | stop | restart | status }
```

Database Server Control

The following is the **cubrid** utility syntax used to control the database server process. The following can be used as *command*; **start** to start the service, **stop** to stop the process, **restart** to restart the process, and **status** to verify the status. In all commands, except **status**, the database name must be assigned as a argument.

```
cubrid server command [<database_name>]
command : { start | stop | restart | status  }
```

Broker Control

The following is the **cubrid** utility syntax used to control the CUBRID broker process. The following can be used as *command*; **start** to start the broker process, **stop** to stop the process, **restart** to restart the process, **status** to verify the status, **on** to start a specific broker and **off** to stop it.

```
cubrid broker command
command : { start | stop | restart | status [<broker_name>] | on <broker_name> | off
<broker_name> | reset <broker_name> | acl {status|reload} <broker_name> }
```

CUBRID HA Control

cubrid heartbeat utility statements for the use of the CUBRID HA feature are as follows: **start** used to execute the HA related process, **stop** used to terminate the process, **reload** used to execute the process according to the HA configuration information by re-reading the information, **deact** used to exclude nodes from CUBRID HA group, and **act** used to include the missing nodes from the group. These can be used as a *command*. For more information, see Utilities of cubrid heartbeat.

```
cubrid heartbeat command
command : { start | stop | reload | deact | act }
```

CUBRID Services

Registering Services

You can register one or more of database server, CUBRID Broker, CUBRID Manager or CUBRID HA as CUBRID services in the database environment configuration file (cubrid.conf). Only a master process is registered by default if you have not registered a specific service by yourself. You can conveniently run, stop or check the status of all related processes at once by using the **cubrid service** utility if they are registered as CUBRID services. The following is an example of registering the database Server and Broker as services in the database environment configuration file, and configuring the **demodb** and **testdb** databases to be started automatically when the CUBRID service starts.

For details about CUBRID HA configuration, see Utilities of cubrid service

The following is an example of configuring cubrid.conf so that database server and broker are registered to service, and demodb and testdb start automatically when CUBRID service starts.

```
# cubrid.conf
…

[service]

# The list of processes to be started automatically by 'cubrid service start' command
# Any combinations are available with server, broker, manager and heartbeat.
service=server,broker

# The list of database servers in all by 'cubrid service start' command.
# This property is effective only when the above 'service' property contains 'server'
keyword.
server=demodb,testdb
```

Starting and Stopping Services

Starting Services

On Linux, after installing CUBRID, enter the following to start a CUBRID service. If no services are registered in the database environment configuration file, only a master process is stopped by default.

On Windows, the following command can be normally executed by a 'SYSTEM' user only. An administrator or general user can run or stop the CUBRID Server by clicking the CUBRID Service tray icon that appears after installing the CUBRID Manager.

```
% cubrid service start
@ cubrid master start
++ cubrid master start: success
```

The following message appears if the master process is already running:

```
% cubrid service start
@ cubrid master start
++ cubrid master is running.
```

The following message appears if the master process fails to start: The following is an example that the service fails to start due to the conflict between the **cubrid_port_id** parameters, which is set in the database environment configuration file (cubrid.conf). In a such case, you can change the port to prevent conflicts. If it fails starting even if no port is occupied by the processes, you should restart it after deleting a /tmp/CUBRID1523 file.

```
% cubrid service start
@ cubrid master start
cub_master: '/tmp/CUBRID1523' file for UNIX domain socket exist.... Operation not
permitted
++ cubrid master start: fail
```

After registering a service as explained in Registering Services, enter the following to start the service. You can see that the master process, database server process, Broker and registered demodb, and testdb all start at the same time.

```
% cubrid service start
@ cubrid master start
++ cubrid master start: success
@ cubrid server start: demodb

This may take a long time depending on the amount of restore works to do.

CUBRID 2008 R4.0

++ cubrid server start: success
@ cubrid server start: testdb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R4.0……

++ cubrid server start: success
@ cubrid broker start
++ cubrid broker start: success
```

Stopping Services

Enter the following to stop a CUBRID service. If no services are registered by the user, only the master process is stopped.

```
% cubrid service stop
@ cubrid master stop
++ cubrid master stop: success
```

Enter the following to stop the registered CUBRID service. You can see that the server process, Broker process and master process as well as demodb and testdb are all stopped.

```
% cubrid service stop
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server stop: testdb
Server testdb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid broker stop
++ cubrid broker stop: success
@ cubrid master stop
++ cubrid master stop: success
```

Restarting Services

Enter the following to restart a CUBRID service. If no services are registered by the user, only the master process is stopped and then restarted.

```
% cubrid service restart
@ cubrid master stop
++ cubrid master stop: success
@ cubrid master start
++ cubrid master start: success
```

Enter the registered CUBRID service as shown below. You can see that the server process, Broker process and master process as well as demodb and testdb are all stopped and then restarted.

```
% cubrid service restart
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server stop: testdb
Server testdb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid broker stop
++ cubrid broker stop: success
@ cubrid master stop
++ cubrid master stop: success
@ cubrid master start
++ cubrid master start: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R4.0……

++ cubrid server start: success
@ cubrid server start: testdb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R4.0……

++ cubrid server start: success
@ cubrid broker start
++ cubrid broker start: success
```

Checking Service Status

Enter the following to check the status of the registered master process and database server.

```
% $ cubrid service status
@ cubrid master status
++ cubrid master is running.
@ cubrid server status

 Server testdb (rel 8.2, pid 31059)
 Server demodb (rel 8.2, pid 30950)

@ cubrid broker status
% query_editor  - cub_cas [15464,40000]
/home1/cubrid22/CUBRID/log/broker//query_editor.access
/home1/cubrid22/CUBRID/log/broker//query_editor.err
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60.00, LONG_QUERY_TIME:60.00, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
----------------------------------------
ID   PID   QPS   LQS PSIZE STATUS
----------------------------------------
 1 15465     0     0 48032 IDLE
 2 15466     0     0 48036 IDLE
 3 15467     0     0 48036 IDLE
 4 15468     0     0 48036 IDLE
 5 15469     0     0 48032 IDLE

@ cubrid manager server status
++ cubrid manager server is not running.
```

The following message appears if the master process has been stopped.

```
% cubrid service status
@ cubrid master status
++ cubrid master is not running.
```

Database Server

Starting and Stopping Database Server

Starting the Database Server

Enter the following to run the demodb server.

```
% cubrid server start demodb
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R4.0

++ cubrid server start: success
```

If you start the demodb server when the master process stops, the master process runs and then the specified database starts automatically.

```
% cubrid server start demodb
@ cubrid master start
++ cubrid master start: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R4.0

++ cubrid server start: success
```

The following message appears if the demodb server is already running.

```
% cubrid server start demodb
@ cubrid server start: demodb
```

```
++ cubrid server 'demodb' is running.
```

cubrid server start only starts cub_server process of the database, regardless of HA mode configuration. If you want to start all HA related processes, you can execute **cubrid heartbeat start**.

Stopping the Database Server

Enter the following to stop the demodb server.

```
% cubrid server stop demodb
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
```

The following message appears if the demodb server has already been stopped.

```
% cubrid server stop demodb
@ cubrid server stop: demodb
++ cubrid server 'demodb' is not running.
```

cubrid server stop only stops cub_server process of the database, regardless of HA mode configuration. The database does not restart, and failover does not occur. If you want to stop all HA related processes, you can execute **cubrid heartbeat stop**.

Restarting the Database Server

Enter the following to restart the demodb server. You can see that the currently running demodb server is stopped and then restarted.

```
% cubrid server restart demodb
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R4.0

++ cubrid server start: success
```

Checking Database Server Status

Enter the following to check the status of the database server. Names of all currently running database servers are displayed.

```
% cubrid server status
@ cubrid server status
 Server testdb (rel 8.2, pid 24465)
 Server demodb (rel 8.2, pid 24342)
```

The following message appears if the master process has been stopped.

```
% cubrid server status
@ cubrid server status
++ cubrid master is not running.
```

Database Server Access Limitation

Description

To limit the brokers and the CSQL interpreters connecting to the database server, set yes for the **access_ip_control** parameter in the **cubrid.conf** file, and input a path of the file in which the list of IP addresses allowed to access the**access_ip_control_file** parameter value is written. You should enter the file path as the absolute path. If you enter the relative path, the system will search the file under the **$CUBRID/conf** directory in Linux and under the **%CUBRID%\conf** directory in Windows.

Configure the **cubrid.conf** file as follows:

```
# cubrid.conf
access_ip_control=yes
access_ip_control_file="/home1/cubrid1/CUBRID/db.access"
```

The format of the **access_ip_control_file** file is as follows:

```
[@<db_name>]
<ip_addr>
…
```

- <db_name> : A database name that allows an access
- <ip_addr> : An IP address allowed to access the database. If the last digit of the address is specified as *, the IP addresses are allowed to access the broker server. You can add multiple lines of <ip_addr> to the next line of one database name.

To configure settings for several database servers, it is possible to specify additional [@<db_name>] and <ip_addr>.

If a value for **access_ip_control** is set to yes and a value for **access_ip_control_file** is not specified, the server will block an access from all IPs and only allow the access from the localhost. If the analysis of **access_ip_control_file** fails due to an incorrect format while the server is running, the server will not run.

The following is an example of **access_ip_control_file**.

```
[@dbname1]
10.10.10.10
10.156.*

[@dbname2]
*

[@dbname3]
192.168.1.15
```

For the above example, the dbname1 database allows the access from the IP of 10.10.10.10 or IPs s starting with 10.156. The dbname2 database allows the access from all IPs. The dbname3 database allows the access from the IP of 192.168.1.15.

For the database which has already been running, you can modify the configuration file or check the currently applied status of configuration by using the following commands.

Syntax

To change the contents of **access_ip_control_file** and apply it to the server, use the following command.

```
cubrid server acl reload <database_name>
```

- *database_name* : A database name

To display the IP configuration for the server which is running, use the following command.

```
cubrid server acl status <database_name>
```

- *database_name* : A database name

Database Server Log

If you access the database server through an IP that is not allowed, the following server error logs will be created in a server error log file.

```
Time: 10/29/10 17:32:42.360 - ERROR *** ERROR CODE = -1022, Tran = 0, CLIENT =
(unknown):(unknown)(-1), EID = 2
Address(10.24.18.66) is not authorized.
```

Note For more information on how to limit an access to the broker server, see Broker Server Access Limitation.

Broker

Starting and Stopping Broker

Enter the following to start the Broker.

```
% cubrid broker start
@ cubrid broker start
++ cubrid broker start: success
```

The following message appears if the Broker is already running.

```
% cubrid broker start
@ cubrid broker start
++ cubrid broker is running.
```

Enter the following to stop the Broker.

```
% cubrid broker stop
@ cubrid broker stop
++ cubrid broker stop: success
```

The following message appears if the Broker has been stopped.

```
% cubrid broker stop
@ cubrid broker stop
++ cubrid broker is not running.
```

Checking Broker Status

Description

By providing various options, the **cubrid broker status** utility allows you to check the status of the Broker, such as the number of completed jobs by each Broker and the number of standby jobs. Take a look at the syntax and its examples.

Syntax

The following is the syntax for monitoring the status of the CUBRID Broker. If *exor* is specified, monitoring of the status of the specified Broker is performed; if omitted, all Brokers registered in the CUBRID Broker environment configuration file (**cubrid_broker.conf**) are monitored.

```
cubrid broker status options [<expr>]
options : [ -b | -f [-l secs] | -q | -t | -s secs ]
```

Options

The following table shows options that can be used together with cubrid broker status.

Options	**Description**
expr	Displays the status of the Broker of which name contains <*expr*>. If this option is not specified, the status of all Brokers is displayed.
-b	Displays the status of the Broker only, excluding the information on the application server (CAS).
-f [**-l** *secs*]	Displays DB and host information accessed by Broker. If it is used with the **-b** option, information on CAS is displayed as well. The **-l** *secs* option is used to specify accumulation period (unit: sec.) when displaying the number of CASs of which status is Waiting or Busy. If the **-l** *secs* option is omitted, one sec. is specified by default.
-q	Displays standby jobs in the job queue.
-t	Displays on screen in tty mode. The output contents can be redirected so that it can be used as a file.
-s	Displays the status of the Broker regularly according to the specified time period. Returns to the command prompt if **q** is entered.

-f	Displays DB and host information where the Broker is connected.

Example

If you do not specify any option and argument to check the status of all Brokers, you will get the following output:

```
% cubrid broker status
@ cubrid broker status
% query_editor - cub_cas [28433,40820] /home/CUBRID/log/broker/query_editor.access
/home/CUBRID/

 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
---------------------------------------------------------------
ID PID   QPS  LQS  PSIZE STATUS
---------------------------------------------------------------
1 28434  0    0    50144 IDLE
2 28435  0    0    50144 IDLE
3 28436  0    0    50144 IDLE
4 28437  0    0    50144 IDLE
5 28438  0    0    50144 IDLE

% broker1  - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
----------------------------------------
ID   PID     QPS   LQS  PSIZE STATUS
----------------------------------------
 1  28444     0     0   50144 IDLE
 2  28445     0     0   50140 IDLE
 3  28446     0     0   50144 IDLE
 4  28447     0     0   50144 IDLE
 5  28448     0     0   50144 IDLE
```

- % query_editor : Broker name
- cub_cas : Type of the CUBRID application server
- [28433, 40820] : Broker process ID and connection port number of the Broker
- /home/CUBRID/log/broker/query_editor.access : Path of the access log file of query_editor
- JOB QUEUE : The number of standby jobs in the job queue
- AUTO_ADD_APPL_SERVER : The value of the AUTO_ADD_APPL_SERVER parameter in **cubrid_broker.conf** is ON, which allows the application server to be added automatically.
- SQL_LOG_MODE : The value of the SQL_LOG parameter in the **cubrid_broker.conf** file is ALL, which allows the SQL log to be recorded.
- LONG_TRANSACTION_TIME : Transaction execution time which determines long-duration transaction. Exceeding 60 seconds is regarded as long-duration transaction.
- LONG_QUERY_TIME : Query execution time which determines long-duration query. Exceeding 60 seconds is regarded as long-duration query.
- SESSION_TIMEOUT : Session timeout value; the value of SESSION_TIMEOUT parameter in the **cubrid_broker.conf** file is 300.
- KEEP_CONNECTION : The value of KEEP_CONNECTION parameter in the **cubrid_broker.conf** file is AUTO, which allows client applications is automatically connected to their server.
- ACCESS_MODE: The Broker action mode; Database manipulation as well as retrieval is allowed in the RW mode.
- ID : Serial number of the application server (CAS) within the Broker
- PID : Application server (CAS) process ID within the Broker
- QPS : The number of queries processed per second
- LQS : The number of long-duration queries processed per second
- PSIZE : Size of the application server process
- STATUS : The current status of the application server (BUSY, IDLE, CLIENT_WAIT, CLOSE_WAIT)

To check the status of the Broker, enter as follows:

```
% cubrid broker status -b
```

```
@ cubrid broker status
  NAME           PID  PORT   AS  JQ      REQ  TPS  QPS  LONG-T  LONG-Q ERR-Q
===========================================================================
* query_editor  4094 30000   5   0        0    0    0    0/60    0/60    0
* broker1       4104 33000   5   0        0    0    0    0/60    0/60    0
```

- NAME : Broker name
- PID : Process ID of the Broker
- PORT : Port number of the Broker
- AS : The number of application servers
- JQ : The number of standby jobs in the job queue
- REQ : The number of client requests processed by the Broker
- TPS : The number of transactions processed per second (calculated only when the option is configured to "-b -s <sec>")
- QPS : The number of queries processed per second (calculated only when the option is configured to "-b -s <sec>")
- LONG-T : The number of transactions which exceed LONG_TRANSACTION_TIME; the value of the LONG_TRANSACTION_TIME parameter
- LONG-Q : The number of queries which exceed LONG_QUERY_TIME; the value of the LONG_QUERY_TIME parameter
- ERR-Q : The number of queries with errors found

Check the status of the Broker whose name contains **broker1** by using the **-q** option, and then enter the following to check the status of the standby jobs in the job queue of the specified Broker. If **broker1** is not entered as an argument, the list of all standby jobs in the job queue of all Brokers is outputted.

```
% cubrid broker status -q broker1
@ cubrid broker status
% broker1  - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
----------------------------------------
ID   PID   QPS   LQS PSIZE STATUS
----------------------------------------
 1 28444     0     0 50144 IDLE
 2 28445     0     0 50140 IDLE
 3 28446     0     0 50144 IDLE
 4 28447     0     0 50144 IDLE
 5 28448     0     0 50144 IDLE
```

Enter the monitoring interval of the Broker whose name contains **broker1** by using the **-s** option, and then enter the following to monitor the status of the Broker regularly. If **broker1** is not entered as an argument, monitoring of the status of all Brokers is performed regularly. If you enter **q**, the monitoring screen returns to the command prompt.

```
% cubrid broker status -s 5 broker1
% broker1  - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
----------------------------------------
ID   PID   QPS   LQS PSIZE STATUS
----------------------------------------
 1 28444     0     0 50144 IDLE
 2 28445     0     0 50140 IDLE
 3 28446     0     0 50144 IDLE
 4 28447     0     0 50144 IDLE
 5 28448     0     0 50144 IDLE
```

Output TPS and QPS information to a file by using the **-t** option. To cancel the output process, press <CTRL+C> to stop the program.

```
% cubrid broker status -b -t -s 1 > log_file
```

Enter the following to monitor the status of all Brokers (including TPS and QPS) regularly by using the **-b** and **-s** options.

```
% cubrid broker status -b -s 1
NAME           PID  PORT  AS  JQ      REQ  TPS  QPS  LONG-T  LONG-Q ERR-Q
```

```
===========================================================================
* query_editor 28433 40820   5   0        0    0    0    0/60    0/60    0
* broker1      28443 40821   5   0        0    0    0    0/60    0/60    0
```

Enter the following to view information of a server/database connected to the Broker, its access time, and the IP addresses connected to CAS by using the **-f** option.

```
$ cubrid broker status -f broker1
@ cubrid broker status
% broker1  - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
------------------------------------------------------------------------------------------
------------------------
ID   PID   QPS   LQS  PSIZE STATUS     LAST ACCESS TIME        DB      HOST   LAST CONNECT
TIME      CLIENT IP
------------------------------------------------------------------------------------------
------------------------
 1  26946   0    0    51168  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
 2  26947   0    0    51172  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
 3  26948   0    0    51172  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
 4  26949   0    0    51172  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
 5  26950   0    0    51172  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
```

The **-b** and the **-f** options are used to display information on AS(T W B Ns-W Ns-B) and CANCELED. The description of each information are as follows:

- T : Total number of CASs being executed
- W : The number of CASs in the state of Waiting
- B : The number of CASs in the state of Busy
- Ns-W : The number of CASs that the client belongs to has been waited for N secs.
- Ns-B : The number of CASs that the client belongs to has been Busy for N secs.
- CANCELED: The number of queries have canceled by user interruption since Broker is started (if it is used with the **-l N** option, it specifies the number of accumulations for N secs).

```
// Adding the -f option upon the execution of Broker state information. The -l option is
used to specify the N value (unit: sec) so that Ns-W and Ns-B can be displayed for
specified N secs.
% cubrid broker status -b -f -l 2
@ cubrid broker status
NAME           PID    PSIZE PORT  AS(T W B 2s-W 2s-B) JQ REQ TPS QPS LONG-T LONG-Q ERR-Q
CANCELED ACCESS_MODE SQL_LOG
==========================================================================================
========================
query_editor 16784 56700 38000    5 0 0 0     0      0  0    0    0   0/60.0 0/60.0
0      0     RW         ALL
```

Broker Server Access Limitation

Description

To limit the client applications accessing the broker, set ON for the **ACCESS_ CONTROL** parameter in the **cubrid_broker.conf** file, and input a name of the file in which the users and the list of databases and IP addresses allowed to access the **ACCESS_CONTROL_FILE** parameter value are written. The default value of the **ACCESS_CONTROL** broker parameter is **OFF**.

The format of **ACCESS_CONTROL_FILE** is as follows:

```
[%<broker_name>]
<db_name>:<db_user>:<ip_list_file>
…
```

- <broker_name> : A broker name. It is the one of broker names specified in **cubrid_broker.conf**.

* <db_name> : A database name. If it is specified as *, all databases are allowed to access the broker server.
* <db_user> : A database user ID. If it is specified as *, all database user IDs are allowed to access the broker server.
* <ip_list_file> : A file name in which the list of IP addresses that are allowed to access the server is written

To configure settings for several broker servers, it is possible to specify additional [%<broker_name>] and <db_name>:<db_user>:<ip_list_file>.

The format of the ip_list_file is as follows:

```
<ip_addr>
…
```

* <ip_addr> : An IP address that is allowed to access the server. If the last digit of the address is specified as *, all IP addresses in that rage are allowed to access the broker server.

If a value for **ACCESS_CONTROL** is set to ON and a value for **ACCESS_CONTROL_FILE** is not specified, the broker will only allow the access requests from the localhost. If the analysis of **ACCESS_CONTROL_FILE** and ip_list_file fails while a broker is running, the broker will only allow the access requests from the localhost.

If the analysis of **ACCESS_CONTROL_FILE** and ip_list_file fails while a broker is running, the broker will not run.

```
# cubrid_broker.conf
[broker]
MASTER_SHM_ID           =30001
ADMIN_LOG_FILE          =log/broker/cubrid_broker.log
ACCESS_CONTROL   =ON
ACCESS_CONTROL_FILE     =/home1/cubrid/access_file.txt
[%QUERY_EDITOR]
SERVICE                 =ON
BROKER_PORT             =38000
......
```

The following is an example of **ACCESS_CONTROL_FILE**. The * symbol represents everything, and you can use it when you want to specify database names, database user IDs and IPs in the IP list file which are allowed to access the broker server.

```
[%QUERY_EDITOR]
dbname1:dbuser1:iplist1.txt
dbname2:*:iplist1.txt
*:dba:iplist1.txt

[%BROKER2]
dbname:dbuser:iplist2.txt

[%BROKER3]
dbname:dbuser:iplist2.txt

[%BROKER4]
dbname:dbuser:iplist2.txt
```

The brokers specified above are QUERY_EDITOR, BROKER2, BROKER3 and BROKER4.

The QUERY_EDITOR broker only allows the following application access requests.

* If you connect to the IP registered in iplist1.txt and log-in to dbname1 with the dbuser1 account.
* If you connect to the IP registered in iplist1.txt and log-in to dbname2.
* If you connect to the IP registered in iplist1.txt and log-in to all databases with the dba user account.

The following is an example of specifying the IPs allowed in ip_list_file.

```
192.168.1.25
192.168.*
10.*
*
```

The descriptions for the IPs specified in the example above are as follows:

* The first line setting allows an access from 192.168.1.25.
* The second line setting allows an access from all IPs starting with 192.168.
* The third line setting allows an access from all IPs starting with 10.
* The fourth line setting allows an access from all IPs.

For the broker which has already been running, you can modify the configuration file or check the currently applied status of configuration by using the following commands.

Syntax

To configure databases, database user IDs and IPs allowed to access the broker and then apply the modified configuration to the server, use the following command.

```
cubrid broker acl reload [<BR_NAME>]
```

- *BR_NAME* : A broker name. If you specify this value, you can apply the changes only to specified brokers. If you omit it, you can apply the changes to all brokers.

To display the databases, database user IDs and IPs that are allowed to access the broker in running on the screen, use the following command.

```
cubrid broker acl status [<BR_NAME>]
```

- *BR_NAME* : A broker name. If you specify the value, you can display the specified broker configuration. If you omit it, you can display all broker configurations.

Broker Log

If you access the broker through an IP that is not allowed, the following logs will be created.

- ACCESS_LOG

```
1 192.10.10.10 - - 1288340944.198 1288340944.198 2010/10/29 17:29:04 ~ 2010/10/29
17:29:04 14942 - -1 db1 dba : rejected
```

- SQL LOG

```
10/29 10:28:57.591 (0) CLIENT IP 192.10.10.10 10/29 10:28:57.592 (0) connect db db1
user dba url jdbc:cubrid:192.10.10.10:30000:db1::: - rejected
```

Note For more information on how to limit an access to the database server, see Database Server Access Limitation.

Managing Specific Broker

Enter the following to start *broker1* only. Here, *broker1* is a broker that has been already configured in the shared memory.

```
% cubrid broker on broker1
```

The following message appears if *broker1* is not configured in the shared memory.

```
% cubrid broker on broker1
Cannot open shared memory
```

Enter the following to stop *broker1* only. Here, you can also remove the service pool of *broker1*.

```
% cubrid broker off broker1
```

Enter the following to restart *broker1*.

```
% cubrid broker restart broker1
```

The broker reset feature disconnects the existing connection and reconnects again when CAS connects to an unwanted database with failover and etc. For example, once the Read Only broker connects to an active server, it does not reconnect to the standby server automatically even though the standby server is available to connect, and you can disconnect the existing connection and reconnect to the standby server again only through the "cubrid broker reset" command.

To reset broker1, input the following:

```
% cubrid broker reset broker1
```

Dynamically Changing Broker Parameters

Description

You can configure the parameters related to running the Broker in the broker environment configuration file (cubrid_broker.conf). For more information, see Parameter by Broker in the "Performance Management Guide." You can also modify some broker parameters temporarily while the Broker is running by using the broker_changer utility. The following broker parameters can be modified dynamically.

- ACCESS_MODE
- ACCESS_LOG
- APPL_SERVER_MAX_SIZE
- KEEP_CONNECTION
- LOG_BACKUP
- SQL_LOG
- SQL_LOG_MAX_SIZE
- STATEMENT_POOLING
- TIME_TO_KILL

Syntax

The syntax for the **broker_changer** utility, which is used to change broker parameters while the Broker is running, is as follows. Enter the name of the currently running Broker for the *broker_name*. The *parameter*s can be used only for dynamically modifiable parameters. The *value* must be specified based on the parameter to be modified. You can specify CAS identifier (*cas_id*) to apply the changes to the specific CAS. *cas_id* is an ID to be output by **cubrid broker status** command.

```
broker_changer  broker_name [cas_id] parameters  value
```

Example 1

Enter the following to configure the SQL_LOG parameter to ON so that SQL logs can be written to the currently running Broker. Such dynamic parameter change is effective only while the Broker is running.

```
% broker_changer query_editor sql_log on
OK
```

Example 2

Enter the following to change Broker's **ACCESS_MODE** to **Read Only** and automatically reset the Broker in HA environment.

```
% broker_changer broker_m access_mode ro
OK
```

Note If you want to control the service using Cubrid utilities in Windows Vista or the later versions of Window, you are recommended to open the command prompt window as an administrator. For more information, see the notes of CUBRID Utilities.

Broker Logs

There are three types of logs that relate to starting the Broker: access, error and SQL logs. Each log can be found in the log directory under the installation directory. You can change the directory where these logs are to be saved through LOG_DIR and ERROR_LOG_DIR parameters of the broker environment configuration file (cubrid_broker.conf).

Checking the Access Log

The access log file records information on the application client and is saved with the name of *broker_name.access*. If the **LOG_BACKUP** parameter is configured to **ON** in the Broker environment configuration file, when the Broker stops properly, the access log file is saved with the date and time that the Broker has stopped. For example, if broker1 stopped at 12:27 P.M. on June 17, 2008, an access file named broker1.access.20080617.1227 is generated in the **log/broker** directory. The following is an example of an access log.

The following is an example and description of an access log file created in the log directory:

```
1 192.168.1.203 - - 972523031.298 972523032.058 2008/06/17 12:27:46~2008/06/17 12:27:47
7118 - -1
2 192.168.1.203 - - 972523052.778 972523052.815 2008/06/17 12:27:47~2008/06/17 12:27:47
7119 ERR 1025
1 192.168.1.203 - - 972523052.778 972523052.815 2008/06/17 12:27:49~2008/06/17 12:27:49
7118 - -1
```

- 1 : ID assigned to the application server of the Broker
- 192.168.1.203 : IP address of the application client
- 972523031.298 : UNIX timestamp value when the client's request processing started
- 2008/06/17 12:27:46 : Time when the client's request processing started
- 972523032.058 : Unix timestamp value when the client's request processing finished
- 2008/06/17 12:27:47 : Time when the client's request processing finished
- 7118 : Process ID of the application server
- -1 : No error occurred during the request processing
- ERR 1025 : Error occurred during the request processing. Error information exists in offset=1025 of the error log file

Checking the Error Log

The error log file records information on errors that occurred during the client's request processing and is stored with the name of *broker_name_app_server_num.err*.

The following is an example and description of an error log:

```
Time: 02/04/09 13:45:17.687 - SYNTAX ERROR *** ERROR CODE = -493, Tran = 1, EID = 38
Syntax: Unknown class "unknown_tbl". select * from unknown_tbl
```

- Time : 02/04/09 13:45:17.687 : Time when the error occurred
- - SYNTAX ERROR : Type of error (e.g. SYNTAX ERROR, ERROR, etc.)
- *** ERROR CODE = -493 : Error code
- Tran = 1 : Transaction ID. -1 indicates that no transaction ID is assigned.
- EID = 38 : Error ID. This ID is used to find the SQL log related to the server or client logs when an error occurs during SQL statement processing.
- Syntax... : Error message (An ellipsis (...) indicates omission.)

Managing the SQL Log

The SQL log file records SQL statements requested by the application client and is stored with the name of *broker_name_app_server_num.sql.log*. The SQL log is generated in the log/broker/sql_log directory when the SQL_LOG parameter is set to ON. Note that the size of the SQL log file to be generated cannot exceed the value set for the SQL_LOG_MAX_SIZE parameter. CUBRID offers the **broker_log_top**, **broker_log_converter**, and **broker_log_runner** utilities to manage SQL logs. Each utility should be executed in a directory where the corresponding SQL log exists.

The following are examples and descriptions of SQL log files:

```
02/04 13:45:17.687 (38) prepare 0 insert into unique_tbl values (1)
02/04 13:45:17.687 (38) prepare srv_h_id 1
02/04 13:45:17.687 (38) execute srv_h_id 1 insert into unique_tbl values (1)
02/04 13:45:17.687 (38) execute error:-670 tuple 0 time 0.000, EID = 39
02/04 13:45:17.687 (0) auto_rollback
02/04 13:45:17.687 (0) auto_rollback 0
*** 0.000
02/04 13:45:17.687 (39) prepare 0 select * from unique_tbl
02/04 13:45:17.687 (39) prepare srv_h_id 1 (PC)
02/04 13:45:17.687 (39) execute srv_h_id 1 select * from unique_tbl
02/04 13:45:17.687 (39) execute 0 tuple 1 time 0.000
02/04 13:45:17.687 (0) auto_commit
02/04 13:45:17.687 (0) auto_commit 0
```

```
*** 0.000
```

* 02/04 13:45:17.687 : Time when the application sent the request
* (39) : Sequence number of the SQL statement group. If prepared statement pooling is used, it is uniquely assigned to each SQL statement in the file.
* prepare 0 : Whether or not it is a prepared statement
* prepare srv_h_id 1 : Prepares the SQL statement as srv_h_id 1.
* (PC) : It is outputted if the data in the plan cache is used.
* SELECT... : SQL statement to be executed. (An ellipsis (...) indicates omission.) For statement pooling, the binding variable of the WHERE clause is represented as a question mark (?).
* Execute 0 tuple 1 time 0.000 : One row is executed. The time spent is 0.000 second.
* auto_commit/auto_rollback : Automatically committed or rolled back. The second auto_commit/auto_rollback is an error code. 0 indicates that the transaction has been completed without an error.

The **broker_log_top** utility analyses the SQL logs which are generated for a specific period. As a result, the information of SQL statements and time execution are outputted in files by order of the longest execution time; the results of SQL statements are stored in **log.top.q** and those of execution time are stored in **log.top.res**, respectively.

The **broker_log_top** utility is useful to analyse the long query. The syntax is as follows:

```
broker_log_top [options] sql_log_file_list
options : {-t | -F  from_date | -T  to_date}
```

The results are outputted in transaction unit if the **-t** option is specified.

SQL statements which are used for a specific period time can be analyzed by using the **-F** and **-T** options. The input format is MM[/DD[hh[:mm[:ss[.msec]]]]], and the part enclosed by [] can be omitted. If you omit the value, it is regarded as that 01 is input for DD, and 0 is input for hh, mm, ss and msec.

```
-- Set the search range to milliseconds
broker_log_top -F "01/19 15:00:25.000" -T "01/19 15:15:25.180" log1.log

-- The part where the time format is omitted is set to 0 by default. This means that -F
"01/19 00:00:00.000" -T "01/20 00:00:00.000" is input.
broker_log_top -F "01/19" -T "01/20" log1.log
```

All logs are outputted by SQL statement if any option is not specified.

The following logs are the results of executing the broker_log_top utility; logs are generated from Nov. 11th to Nov. 12th, and it is displayed in the order of the longest execution of SQL statements. Each month and day are separated by a slash (/) when specifying period. Note that "*.sql.log" is not recognized so the SQL logs should separated by a white space on Windows.

```
--Execution broker_log_top on Linux
% broker_log_top -F "11/11" -T "11/12" -t *.sql.log

query_editor_1.sql.log
query_editor_2.sql.log
query_editor_3.sql.log
query_editor_4.sql.log
query_editor_5.sql.log

--Executing broker_log_top on Windows
% broker_log_top -F "11/11" -T "11/12" -t query_editor_1.sql.log query_editor_2.sql.log
query_editor_3.sql.log query_editor_4.sql.log query_editor_5.sql.log
```

The log.top.q and log.top.res files are generated in the same directory where the analyzed logs are stored when executing the example above; In the log.top.q file, you can view each SQL statement, and its line number. In the log.top.res, you can the minimum, maximum and avg. time, and the number of execution queries for each SQL statement.

```
--log.top.q file
[Q1]-------------------------------------------
broker1_6.sql.log:137734
11/11 18:17:59.396 (27754) execute_all srv_h_id 34 select a.int_col, b.var_col from
dml_v_view_6 a, dml_v_view_6 b, dml_v_view_6 c , dml_v_view_6 d, dml_v_view_6 e where
a.int_col=b.int_col and b.int_col=c.int_col and c.int_col=d.int_col and
d.int_col=e.int_col order by 1,2;
```

```
11/11 18:18:58.378 (27754) execute_all 0 tuple 497664 time 58.982
.
.
[Q4]-------------------------------------------
broker1_100.sql.log:142068
11/11 18:12:38.387 (27268) execute_all srv_h_id 798 drop table list_test;
11/11 18:13:08.856 (27268) execute_all 0 tuple 0 time 30.469

-- log.top.res file
max           min      avg     cnt(err)
-----------------------------------------------------
[Q1]        58.982    30.371    44.676    2 (0)
[Q2]        49.556    24.023    32.688    6 (0)
[Q3]        35.548    25.650    30.599    2 (0)
[Q4]        30.469    0.001     0.103  1050 (0)
```

To store SQL logs created in log/broker/sql_log under the installation directory to a separate file, the **broker_log_converter** utility is executed. The syntax of the **broker_log_converter** utility is as follows: This example saves queries stored in the query_editor_1.sql.log file to the query_convert.in file.

```
broker_log_converter  SQL_log_file  output_file
```

The following example shows that the query in the query_editor_1.sql.log file is converted into the query_convert.in file.

```
% broker_log_converter query_editor_1.sql.log query_convert.in
```

To re-execute queries saved in the query file which has been created by the **broker_log_converter** utility, the **broker_log_runner** utility is executed. The syntax of the **broker_log_runner** utility is as follows: This example re-executes queries saved in the query_convert.in in demodb. It is assumed that the IP address of the Broker is 192.168.1.10 and its port number is 30,000.

```
broker_log_runner options input_file
options : -I cas_ip -P cas_port  -d dbname  [-u dbuser [-p dbpasswd ]]  [-t num_thread] [-
r repeat_count] [Q] [ -o result_file]
```

broker_log_runner Utility Options

Option	Description
-I *broker_ip*	IP address or host name of the CUBRID Broker
-P *broker_port*	Port number of the CUBRID Broker
-d *dbname*	Name of the database against which queries are to be executed
-u *dbuser*	Database user name (default value : public)
-p *dbpasswd*	Database password
-t *numthread*	The number of threads (default value : 1)
-r *repeat_count*	The number of times that the query is to be executed (default value : 1)
-Q	Stores the query plan in *result_file*. Name of the file where execution results are to be stored
-o *result_file*	Name of the file where execution results are to be stored

```
% broker_log_runner -I 192.168.1.10  -P 30000 -d demodb -t 2 query_convert.in
cas_ip = 192.168.1.10
cas_port = 30000
num_thread = 2
repeat = 1
dbname = demodb
dbuser = public
dbpasswd =
exec_time : 0.001
exec_time : 0.000
0.000500 0.000500 -

% broker_log_runner -I 192.168.1.10  -P 30000 -d demodb -o result -Q query_convert.in
 …
```

```
%cat result.0
-------------- query -----------------
SELECT * FROM athlete where code=10099;

cci_execute:1
---------- query plan --------------
Join graph segments (f indicates final):
seg[0]: [0]
seg[1]: code[0] (f)
seg[2]: name[0] (f)
seg[3]: gender[0] (f)
seg[4]: nation_code[0] (f)
seg[5]: event[0] (f)
Join graph nodes:
node[0]: athlete athlete(6677/107) (sargs 0)
Join graph terms:
term[0]: (athlete.code=10099) (sel 0.000149768) (sarg term) (not-join eligible) (indexable
code[0]) (loc 0)

Query plan:

iscan
    class: athlete node[0]
    index: pk_athlete_code term[0]
    cost:  fixed 0(0.0/0.0) var 0(0.0/0.0) card 1

Query stmt:

select athlete.code, athlete.[name], athlete.gender, athlete.nation_code, athlete.event
from athlete athlete where (athlete.code= ?:0 )


---------- query result --------------
10099|Andersson Magnus|M|SWE|Handball|
-- 1 rows ----------------------------
```

CUBRID Manager Server

Starting and Stopping CUBRID Manager

Starting the CUBRID Manager

Enter the following to run the CUBRID Manager Server.

```
% cubrid manager start
```

The following message appears if the CUBRID Manager server is already running.

```
% cubrid manager start
@ cubrid manager server start
++ cubrid manager server is running.
```

Stopping the CUBRID Manager

Enter the following to stop the CUBRID Manager server.

```
% cubrid manager stop
@ cubrid manager server stop
++ cubrid manager server stop: success
```

CUBRID Manager Server Log

CUBRID Manager Server-related logs are stored in log/manager directory under the installation directory. They are stored as one of the following four types of files depending on the process of the Manager Server.

- cub_auto.access.log : Access log of a client that logged into and out of the Manager Server successfully
- cub_auto.error.log : Access log of a client that failed to log into or out of the Manager Server
- cub_js.access.log : Log of the jobs processed by the Manager Server
- cub_js.error.log : Error log that occurred while the Manager Server is processing jobs

Database Administration

How to Use the CUBRID Administration Utilities (Syntax)

The following shows how to use the CUBRID management utilities.

```
cubrid utility_name
utility_name :
  createdb [option] <database_name>   --- Creating a database
  deletedb [option] <database_name>   --- Deleting a database
  installdb [option] <database-name>   --- Installing a database
  renamedb [option] <source-database-name> <target-database-name>  --- Renaming a database
  copydb [option] <source-database-name> <target-database-name>  --- Copying a database
  backupdb [option] <database-name>  --- Backing up a database
  restoredb [option] <database-name>  --- Restoring a database
  addvoldb [option] <database-name> number-of-pages  --- Adding a database volume file
  spacedb [option] <database-name>  --- Displaying details of database space
  lockdb [option] <database-name>  --- Displaying details of database lock
  killtran [option] <database-name>  --- Removing transactions
  optimizedb [option] <database-name>  --- Updating database statistics
  statdump [option] <database-name>  --- Outputting statistic information of database
server execution
  compactdb [option] <database-name>  --- Optimizing space by freeing unused space
  diagdb [option] <database-name>  --- Displaying internal information
  checkdb [option] <database-name>  --- Checking database consistency
  alterdbhost [option] <database-name>  --- Altering database host
  plandump [option] <database-name>  --- Displaying details of the query plan
  loaddb [option] <database-name>  --- Loading data and schema
  unloaddb [option] <database-name>  --- Unloading data and schema
  paramdump [option] <database-name>  --- Checking out the parameter values configured in
a database
  changemode [option] <database-name>  --- Displaying or changing the server HA mode
  copylogdb [option] <database-name>  --- Multiplating transaction logs to configure HA
  applylogdb [option] <database-name>  --- Reading and applying replication logs from
transaction logs to configure HA
```

Database Users

A CUBRID database user can have members with the same authorization. If authorization **A** is granted to a user, the same authorization is also granted to all members belonging to the user. A database user and its members are called a "group."
CUBRID provides **DBA** and **PUBLIC** users by default.

- **DBA** can access every object in the database, that is, it has authorization at the highest level. Only **DBA** has sufficient authorization to add, alter and delete the database users.
- All users including **DBA** are members of **PUBLIC**. Therefore, all database users have the authorization granted to **PUBLIC**. For example, if authorization **B** is added to **PUBLIC** group, all database members will automatically have the **B** authorization.

databases.txt File

Description

CUBRID saves information on the locations of all existing databases in the **databases.txt** file. This file is called the "database location file." A database location file is used when CUBRID executes utilities for creating, renaming, deleting or replicating databases; it is also used when CUBRID runs each database. By default, this file is located in the **databases** directory under the installation directory. The directory is located through the environment variable **CUBRID_DATABASES**.

Syntax

```
db_name db_directory server_host logfile_directory
```

The format of each line of a database location file is the same as defined by the above syntax; it contains information on the database name, database path, server host and the path to the log files. The following is an example of checking the contents of a database location file.

```
% more databases.txt
dist_testdb /home1/user/CUBRID/bin d85007 /home1/user/CUBRID/bin
dist_demodb /home1/user/CUBRID/bin d85007 /home1/user/CUBRID/bin
testdb /home1/user/CUBRID/databases/testdb d85007 /home1/user/CUBRID/databases/testdb
demodb /home1/user/CUBRID/databases/demodb d85007 /home1/user/CUBRID/databases/demodb
```

By default, the database location file is stored in the **databases** directory under the installation directory. You can change the default directory by modifying the value of the **CUBRID_DATABASES** environment variable. The path to the database location file must be valid so that the **cubrid** utility for database management can access the file properly. You must enter the directory path correctly and check if you have write permission on the file. The following is an example of checking the value configured in the **CUBRID_DATABASES** parameter.

```
% set | grep CUBRID_DATABASES
CUBRID_DATABASES=/home1/user/CUBRID/databases
```

An error occurs if an invalid directory path is set in the **CUBRID_DATABASES** environment variable. If the directory path is valid but the database location file does not exist, a new location information file is created. If the **CUBRID_DATABASES** environment variable has not been configured at all, CUBRID retrieves the location information file in the current working directory.

Creating Database

Descripton

The **cubrid createdb** utility creates databases and initializes them with the built-in CUBRID system tables. It can also define initial users to be authorized in the database and specify the locations of the logs and databases. Generally, the **cubrid createdb** utility is used only by DBA.

Syntax

```
cubrid createdb options database_name
options :
[--db-volume-size=size] [--db-page-size=size] [--log-volume-size=size] [--log-page-
size=size] [--comment=comment] [{-F |--file-path=}path] [{-L |--log-path=}path] [{-B |--
lob-base-path=}path] [--server-name=host] [-r|--replace] [--more-volume-file=file] [--
user-definition-file=file] [--csql-initialization-file=file] [{-o |--output-file=}file] [-
v|--verbose]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **createdb** : A command used to create a new database.
- *options* : A short option starts with a single dash (-) while a full name option starts with a double dash (--).
- *database_name* : Specifies a unique name for the database to be created, without including the path name to the directory where the database will be created. If the specified database name is the same as that of an existing database name, CUBRID halts creation of the database to protect existing files.

Option

The following table shows options that can be used with **cubrid createdb**. Options are case sensitive.

Option	Description
--db-volume-size	Specifies the size of the database volume that will be created first in bytes. Default value : A value of *db_volume_size*, the system parameter
--db-page-size	Specifies the database page size in bytes. Default value : 16K
--log-volume-size	Specifies the log volume size in bytes.
--log-page-size	Specifies the page size of log volume in bytes. Default value : Database page size

--comment	Adds information on the database to be created in the form of a comment.
-F --file-path	Specifies the directory path where the database will be created. Default value : Current working directory
-L --log-path	Specifies the directory path where log files will be stored. Default value : A directory path specified with the **-F** option
-B --lob-base-path	Specifies the directory path where LOB data files will be stored. Default value : <location of database volumns created>/lob directory
--server-name	Specifies the name of the server host to connect to. Default value : localhost
-r --replace	Allows overwriting if the name of the database to be created is the same as that of an existing database. Default value : Deactivated
--more-volume-file	Specifies the file that includes the specifications for creating an additional volume of the database.
--user-definition-file	Specifies the file that includes user definitions.
--csql-initialization-file	Specifies the file for csql initialization.
-o --output-file	Specifies the file where output messages concerning database creation are stored.
-v --verbose	Displays detailed messages to the screen concerning database creation. Default value : Deactivated

Size of the first database volume (--db-volume-size)

The **--db-volume-size** option is used to specify the size of the database volume that will be created first. The default value is the value of the system parameter **db_volume_size**, and the minimum value is 20M. You can set units as K, M, G and T, which stand for KB(kilobytes), MB(megabytes), GB(gigabytes) and TB(terabytes), respectively. If you omit the unit, bytes will be applied.

The following example shows creating a database named testdb and assigning 256MB to its first volume.

```
cubrid createdb --db-volume-size=512M testdb
```

Database page size (--db-page-size)

The **--db-page-size** option is used to specify the size of the database page to be one of 4K(4KB), 8K(8KB), and 16K(16KB). The default vaule is **16K**. If any number besides these is specified, the system rounds up the number. If a number which is greater than 16K, 16K is used.

The following example shows creating a database named testdb and setting its page size 16KB.

```
cubrid createdb --db-page-size=16K testdb
```

Log volume size (--log-volume-size)

The **--log-volume-size** option is used to specify the size of the database log volume. The default value is the databse volume size, and the minimum value is 20M. You can set units as K, M, G and T, which stand for KB(kilobytes),MB (megabytes), GB(gigabytes) and TB(terabytes), respectively. If you omit the unit, bytes will be applied.

The following example shows creating a database named testdb and assigning 256MB to its log volume.

```
cubrid createdb --log-volume-size=256M testdb
```

Log page size (--log-page-size)

The **--log-page-size** option specifies the size of the log volume page to be one of 4K(4KB), 8K(8KB), and 16K(16KB). The default value is the size of the data page. If any number besides these is specified, the system rounds up the number. If a number which is greater than 16K, 16K is used.

The following example shows creating a database named testdb and setting its log volume page size 8KB.

```
cubrid createdb --log-page-size=8K testdb
```

Comment (--comment)

The **--comment** option is used to specify a comment to be included in the database volume header. If the character string contains spaces, the comment must be enclosed in double quotes.

The following example shows creating a database named testdb and adding a related comment to the database volume.

```
cubrid createdb --comment "a new database for study" testdb
```

Database directory path (-F)

The **-F** option is used to specify the absolute path to a directory where the new database will be created. If the **-F** option is not specified, the new database is created in the current working directory.

The following example shows creating a database named testdb in the directory /dbtemp/new_db.

```
cubrid createdb -F "/dbtemp/new_db/" testdb
```

Log file directory path (-L)

The **-L** option is used to specify the absolute path to the directory where database log files are created. If the **-L** option is not specified, log files are created in the directory specified by the **-F** option. If neither **-F** nor **-L** option is specified, database log files are created in the current working directory.

The following example shows creating a database named testdb in the directory /dbtemp/newdb and log files in the directory /dbtemp/db_log.

```
cubrid createdb -F "/dbtemp/new_db/" -L "/dbtemp/db_log/" testdb
```

LOB data file directory (--lob-base-path)

The **--lob-base-path** option is used to specify a directory where LOB data files are stored when BLOB/CLOB data is used. If the **--lob-base-path** option is not specified, LOB data files are store in "<location of database volumns created>/lob" directory.

The following example shows creating a database called testdb in the working directory and specifying "/home/data1" of local file system as a location of LOB data files.

```
cubrid createdb --lob-base-path "file:/home1/data1" testdb
```

Server host name (--server-name)

The **--server-name** option is used to specify that the server for a certain database will be running on a specified host when a client / server version of CUBRID is used. The information on the server host specified with this option is written in the database location file (**databases.txt**). If this option is not specified, the default value is the current localhost.

The following example shows creating and registering a database named testdb on the *aa_host* host.

```
cubrid createdb --server-name aa_host testdb
```

Overwrite (-r)

The **-r** option is used to create a new database and overwrite an existing database if one with the same name exists. If the **-r** option is not specified, database creation is halted when this occurs.

The following example shows creating a new testdb database which overwrites the existing database with the same name.

```
cubrid createdb -r testdb
```

Adding a database volume (--more-volume-file)

The **--more-volume-file** option creates an additional volume based on the specification contained in the file specified by the option. The volume is created in the same directory where the database is created. Instead of using this option, you can add a volume by using the **cubrid addvoldb** utility.

The following example shows creating a database named testdb as well as an additional volume based on the specification stored in the *vol_info.txt* file.

```
cubrid createdb --more-volume-file vol_info.txt testdb
```

The following is a specification of the additional volume contained in the *vol_info.txt* file. The specification of each volume must be written on a single line.

```
#xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# NAME volname COMMENTS volcmnts PURPOSE volpurp NPAGES volnpgs
NAME data_v1 COMMENTS "Data information volume" PURPOSE data NPAGES 1000
NAME data_v2 COMMENTS "Data information volume" PURPOSE data NPAGES 1000
NAME data_v3 PURPOSE data NPAGES 1000
NAME index_v1 COMMENTS "Index information volume" PURPOSE index NPAGES 500
NAME temp_v1 COMMENTS "Temporary information volume" PURPOSE temp NPAGES 500
NAME generic_v1 COMMENTS "Generic information volume" PURPOSE generic NPAGES 500
#xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

As shown in the example, the specification of each volume is composed of followings.

```
NAME volname COMMENTS volcmnts PURPOSE volpurp NPAGES volnpgs
```

- **NAME** *volname* : *volname* is the name of the volume to be created. It must follow the UNIX file name conventions and be a simple name not including the directory path. The specification of a volume name can be omitted. If it is, the "database name to be created by the system_volume identifier" becomes the volume name.
- **COMMENTS** *volcmnts* : *volcmnts* is a comment to be written in the volume header and contains information on the additional volume to be created. The specification of the comment on a volume can also be omitted.
- **PURPOSE** *volpurp* : *volpurp* must be one of the types: **data**, **index**, **temp**, and **generic**, with the purpose of saving volumes. The specification of the purpose of a volume can be omitted in which case the default value is **generic**.
- **NPAGES** *volnpgs* : *volnpgs* is the number of pages of the additional volume to be created. The specification of the number of pages of the volume cannot be omitted; it must be specified.

User information file (--user-definition-file)

The **--user-definition-file** option is used to add users who have access to the database to be created. It adds a user based on the specification contained in the user information file specified by the parameter. Instead of using the **--user-definition-file** option, you can add a user by using the Managing USER statement.

The following example shows creating a database named testdb and adding users to testdb based on the user information defined in the user_info.txt file.

```
cubrid createdb --user-definition-file user_info.txt testdb
```

The syntax of a user information file is as follows:

```
USER user_name [ groups_clause | members_clause ]|
groups_clause:
 [ GROUPS group_name [ { group_name }... ] ]
members_clause:
 [ MEMBERS member_name [ { member_name... } ] ]
```

- The *user_name* is the name of the user who has access to the database. It must not include spaces.
- The **GROUPS** clause is optional. The *group_name* is the upper level group that contains the *user_name*. Here, the *group_name* can be multiply specified and must be defined as **USER** in advance.
- The **MEMBERS** clause is optional. The *member_name* is the name of the lower level member that belongs to the *user_name*. Here, the *member_name* can be multiply specified and must be defined as **USER** in advance.

Comments can be used in a user information file. A comment line must begin with a hyphen (-). Blank lines are ignored.

The following example is a user information file that defines the group sedan to include grandeur and sonata, the group suv to include tuscan, and the group hatchback to include i30. The name of the user information file is user_info.txt.

```
--
-- Example 1 of a user information file
--
USER sedan
USER suv
USER hatchback
USER grandeur GROUPS sedan
USER sonata GROUPS sedan
```

```
USER tuscan GROUPS suv
USER i30 GROUPS hatchback
```

The following file defines the same user relationship as the one above, except that it uses the **MEMBERS** clause.

```
--
-- Example 2 of a user information file
--
USER grandeur
USER sonata
USER tuscan
USER i30
USER sedan MEMBERS sonata grandeur
USER suv MEMBERS tuscan
USER hatchback MEMBERS i30
```

File containing CSQL statements (--csql-initialization-file)

The **--csql-initialization-file** option executes an SQL statement on the database to be created by using the CSQL Interpreter. A schema can be created based on the SQL statement contained in the file specified by the parameter.

The following example shows creating a database named testdb and executing the SQL statement defined in table_schema.sql through the CSQL Interpreter.

```
cubrid createdb --csql-initialization-file table_schema.sql testdb
```

Saving output messages to a file (-o)

The **-o** option is used to save messages related to the database creation to the file given as a parameter. The file is created in the same directory where the database was created. If the **-o** option is not specified, messages are displayed on the console screen. The **-o** option allows you to use information on the creation of a certain database by saving messages, generated during the database creation, to a specified file.

The following example shows creating a database named testdb and saving the output of the utility to the db_output file instead of displaying it on the console screen.

```
cubrid createdb -o db_output testdb
```

Verbose output (-v)

The **-v** option is used to output all information on the database creation operation onto the screen. Like the **-o** option, this option is useful in checking information related to the creation of a specific database. Therefore, if you specify the **-v** option together with the **-o** option, you can save the output messages in the file given as a parameter; the messages contain the operation information about the **cubrid createdb** utility and database creation process.

The following example shows creating a database named testdb and outputting detailed information on the operation onto the screen.

```
cubrid createdb -v testdb
```

Adding Database Volume

Syntax

```
cubrid addvoldb options database_name
options :
[--db-volume-size=size] [{-n |--volume_name=}name] [{-F |--file-path=}path] [--
comment=comment] [-p|--purpose] [-S|--SA-mode|-C|--CS-mode]
```

- **cubrid** : An integrated utility for CUBRID service and database management.
- **addvoldb** : A command that adds a specified number of pages of the new volume to a specified database.
- *options* : A short option starts with a single dash (-) while a full name option starts with a double dash (--).
- *database_name* : Specifies the name of the database to which a volume is to be added without including the path name to the directory where the database is to be created.

Option

The following table shows options that can be used with **cubrid addvoldb** utility.

Option	Description
--db-volume-size	Specifies the database volume size in bytes. Default value : A value of *db_volume_size*, the system parameter
-n **--volume-name**	Specifies the name of the database volume to be added. Default value : A value in the format of *dbname_number*, configured by the system
-F **--file-path**	Specifies the directory path where the database volume to be added will be created. Default value : A value of **volume_extension_path**, the system parameter
--comment	Inserts a comment about the database volume to be added.
-p **--purpose**	Specifies the purpose of the database volume to be added. Default value : Generic volume
-S **--SA-mode**	Adds the database volume in standalone mode.
-C **--CS-mode**	Adds the database volume in client/server mode.

Size of the extended volume (--db-volume-size)

--db-volume-size is an option that specifies the size of the volume to be added to a specified database. If the **--db-volume-size** option is omitted, the value of the system parameter **db_volume_size** is used by default. You can set units as K, M, G and T, which stand for KB(kilobytes), MB(megabytes), GB(gigabytes) and TB(terabytes), respectively. If you omit the unit, bytes will be applied.

The following example shows adding a volume for which 256MB are assigned to the testdb database.

```
cubrid addvoldb -p data --db-volume-size=256M testdb
```

Name of the extended volume (-n)

-n is an option that specifies the name of the volume to be added to a specified database. The volume name must follow the file name protocol of the operating system and be a simple one without including the directory path or spaces. If the **-n** option is omitted, the name of the volume to be added is configured by the system automatically as "database name_volume identifier." For example, if the database name is testdb, the volume name testdb_x001 is automatically configured.

The following example shows adding a volume for which 1000 pages are assigned to the testdb database in standalone mode. The volume name testdb_v1 will be created.

```
cubrid addvoldb -S -n testdb_v1 testdb 1000
```

Path of the extended volume (-F)

The **-F** option is used to specify the directory path where the volume to be added will be stored. If the **-F** option is omitted, the value of the system parameter **volume_extension_path** is used by default.

The following example shows adding a volume for which 1000 pages are assigned to the testdb database in standalone mode. The added volume is created in the /dbtemp/addvol directory. Because the **-n** option is not specified for the volume name, the volume name testdb_x001 will be created.

```
cubrid addvoldb -S -F /dbtemp/addvol/ testdb 1000
```

Comment about the added volume (--comment)

The **--comment** option is used to facilitate to retrieve information on the added volume by adding such information in the form of comments. It is recommended that the contents of a comment include the name of **DBA** who adds the volume, or the purpose of adding the volume. The comment must be enclosed in double quotes.

The following example shows adding a volume for which 1000 pages are assigned to the testdb database in standalone mode and inserts a comment about the volume.

```
cubrid addvoldb -S --comment "data volume added_cheolsoo kim" testdb 1000
```

Purpose of the volume (-p)

The **-p** option is used to specify the purpose of the volume to be added. The reason for specifying the purpose of the volume is to improve the I/O performance by storing volumes separately on different disk drives according to their purpose. Parameter values that can be used for the **-p** option are **data**, **index**, **temp** and **generic**. The default value is **generic**. For the purpose of each volume, see "Database Volume Structure."

The following example shows adding a volume for which 1000 pages are assigned to the testdb database in standalone mode.

```
cubrid addvoldb -S -p index testdb 1000
```

Standalone mode (-S)

The **-S** option is used to access the database in standalone mode without running the server process. This option has no parameter. If the **-S** option is not specified, the system assumes to be in client/server mode.

```
cubrid addvoldb -S testdb 1000
```

Client/server mode (-C)

The **-C** option is used to access the database in client/server mode by running the server and the client separately. There is no parameter. Even when the **-C** option is not specified, the system assumes to be in client/server mode by default. If the **-S** or **-C** option is not specified and the environment variable **CUBRID_MODE** is not defined, the system assumes to be in client/server mode.

```
cubrid addvoldb -C -testdb 1000
```

Deleting Database

Description

The **cubrid deletedb** utility is used to delete a database. You must use the **cubrid deletedb** utility to delete a database, instead of using the file deletion commands of the operating system; a database consists of a few interdependent files. The **cubrid deletedb** utility also deletes the information on the database from the database location file (**databases.txt**). The **cubrid deletedb** utility must be run offline, that is, in standalone mode when nobody is using the database.

Syntax

```
cubrid deletedb options database_name
options : [{-o|--output-file=} file]  [-d|--delete-backup]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **deletedb** : A command to delete a database, its related data, logs and all backup files. It can be executed successfully only when the database is in a stopped state.
- *options* : **-o** and **-d** options are provided.
- *database_name* : Specifies the name of the database to be deleted without including the path name.

Option

Saving output messages (-o **or** --output-file**)**

The following example shows deleting testdb and writes output messages to the file specified by using the **-o** option.

```
cubrid deletedb -o deleted_db.out testdb
```

The **cubrid deletedb** utility also deletes the database information contained in the database location file (**databases.txt**). The following message appears if you enter a utility that tries to delete a non-existing database.

```
cubrid deletedb testdb
```

```
Database "testdb" is unknown, or the file "databases.txt" cannot be accessed.
```

Deleting backup files simultaneously (-d **or** --delete-backup**)**

The following example shows deleting testdb and its backup volumes and backup information files simultaneously by using the **-d** option. If the **-d** option is not specified, backup volume and backup information files are not deleted.

```
cubrid deletedb -d testdb
```

Renaming Database

Description

The **cubrid renamedb** utility renames a database. The names of information volumes, log volumes and control files are also renamed to conform to the new database one.

The **cubrid alterdbhost** utility configures or changes the host name of the specified database. It changes the host name configuration in the **databases.txt** file.

Syntax

```
cubrid renamedb options src_database_name dest_database_name
options : [{-E | --extended-volumn-path=}path ] [ {-i | --control-file=} file ] [-d | --
delete-backup]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **renamedb** : A command that changes the existing name of a database to a new one. It executes successfully only when the database is in a stopped state. The names of related information volumes, log volumes and control files are also changed to new ones accordingly.
- *options* : The **-E**, **-i** and **-d** options are supported. For details about each option, see its description and the examples.
- *src_database_name* : The name of the existing database to be renamed. The path name to the directory where the database is to be created must not be included.
- *dest_database_name* : The new name of the database. It must not be the same as that of an existing database. The path name to the directory where the database is to be created must not be included.

Option

Saving the renamed extended volume to a new directory (-E or --extended-volume-path)

The following example shows renaming an extended volume created in a specific directory path (e.g. /dbtemp/addvol/) with a **-E** option, and then moves the volume to a new directory. The **-E** option is used to specify a new directory path (e.g. /dbtemp/newaddvols/) where the renamed extended volume will be moved. If the **-E** option is not specified, the extended volume is only renamed in the existing path without being moved. If a directory path outside the disk partition of the existing database volume or an invalid one is specified, the rename operation is not executed. This option cannot be used together with the **-i** option.

```
cubrid renamedb -E /dbtemp/newaddvols/ testdb testdb_1
```

Specifying the input file where the directory information is stored (-i or --control-file)

The following example shows specifying an input file which saves directory information with an **-i** option, to assign different directories as well as to change database names for each volume and file at once. The **-i** option cannot be used together with the **-E** option.

```
cubrid renamedb -i rename_path testdb testdb_1
```

The followings are the syntax and example of a file that contains the name of each volume, the current directory path and the directory path where renamed volumes will be saved.

```
volid   source_fullvolname   dest_fullvolname
```

- *volid* : An integer that is used to identify each volume. It can be checked in the database volume control file (database_name_vinf).
- *source_fullvolname* : The current directory path to each volume.

- *dest_fullvolname* : The target directory path where renamed volumes will be moved. If the target directory path is invalid, the database rename operation is not executed.

```
  -5  /home1/user/testdb_vinf       /home1/CUBRID/databases/testdb_1_vinf
  -4  /home1/user/testdb_lginf      /home1/CUBRID/databases/testdb_1_lginf
  -3  /home1/user/testdb_bkvinf     /home1/CUBRID/databases/testdb_1_bkvinf
  -2  /home1/user/testdb_lgat       /home1/CUBRID/databases/testdb_1_lgat
   0  /home1/user/testdb            /home1/CUBRID/databases/testdb_1
   1  /home1/user/backup/testdb_x001/home1/CUBRID/databases/backup/testdb_1_x001
```

Deleting and renaming backup files simultaneously (-d or --delete-backup)

By using the **-d** option, the following example shows renaming the testdb database and at the same time forcefully deletes all backup volumes and backup information files that are in the same location as testdb. Note that you cannot use the backup files with the old names once the database is renamed. If the **-d** option is not specified, backup volumes and backup information files are not deleted.

```
cubrid renamedb -d testdb testdb_1
```

Copying/Moving Database

Description

The **cubrid copydb** utility copy or move a database to another location. As arguments, source and target name of database must be given. A target database name must be different from a source database name. Wh the target name argument is specified, the location of target database name is registered in the **databases.txt** file. The **cubrid copydb** utility can be executed only offline (that is, state of a source database stop).

Syntax

```
cubrid copydb [OPTION] src-database-name dest-database-name

options : [{--server-name=}host] [{-F | --file-path=} database_path ] [ {-L | --log-path=}
log_path ] [{-B | --lob-base-path=} lob_file_path] [{-E | --extended-volume-path=}
path ][{-i, --control-file=}FILE ] [ -r | --replace ] [ -d | --delete-source ] [ --copy-
lob-path ]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **copyd** : A command that copy or move a database from one to another location.
- *options* : For details about each option, see its description and the examples. If options are omitted, a target database is copied into the same directory of a source database.
- *src_database_name* : The names of source and target databases to be copied or moved.
- *dest_database_name* : A new (target) database name.

Option

Registering the host name (--server-name)

The following example shows specifying a host name of new database. The host name is registered in the **databases.txt** file. If this option is omitted, a local host is registered.

```
cubrid copydb --server-name=cub_server1 demodb new_demodb
```

Storing a new database volume in a specific directory (-F or --file-path)

The following example shows specifying a specific directory path where a new database volume is stored with an **-F** option. It represents specifying an absolute path. If the specified directory does not exist, an error is outputted. If this option is omitted, a new database volume is created in the current working directory. And this information is specified in **vol-path** of the **databases.txt** file.

```
cubrid copydb -F /home/usr/CUBRID/databases demodb new_demodb
```

Storing a new database log volume in a specific directory (-L or --log-path)

The following example shows specifying a specific directory path where a new database volume is stored with an **-L** option. It represents specifying an absolute path. If the specified directory does not exist, an error is outputted. If this option is omitted, a new database volume is created in the current working directory. And this information is specified in **log-path** of the **databases.txt** file.

```
cubrid copydb -L /home/usr/CUBRID/databases/logs demodb new_demodb
```

Storing a new database extended volume in a specific directory (-E or --extended-volume-path)

The following example shows specifying a specific directory path where a new database extended volume is stored with an **-E**. If this option is omitted, a new database extended volume is created in the location of a new database volume or in the registered path of controlling file. The **-i** option cannot be used with this option.

```
cubrid copydb -E home/usr/CUBRID/databases/extvols demodb new_demodb
```

Specifying an input file where directory path information is stored (-i or --control file)

The following example shows specifying an input file where a new directory path information and a source volume are stored to copy or move multiple volumes into a different directory, respectively. The **-i** option cannot be used with the **-E** option. An input file named copy_path is specified in the example below.

```
cubrid copydb -i copy_path demodb new_demodb
```

The following is an exmaple of input file that contains each volume name, current directory path, and new directory and volume names.

```
# volid source_fullvolname dest_fullvolname
0 /usr/databases/demodb /drive1/usr/databases/new_demodb
1 /usr/databases/demodb_data1 /drive1/usr/databases/new_demodb new_data1
2 /usr/databases/ext/demodb index1 /drive2//usr/databases/new_demodb new_index1
3 /usr/ databases/ext/demodb index2 /drive2/usr/databases/new_demodb new_index2
```

- *volid* : An integer that is used to identify each volume. It can be checked in the database volume control file (**database_name_vinf**).
- *source_fullvolname* : The current directory path to each source database volume.
- *dest_fullvolname* : The target directory path where new volumes will be stored. You should specify a vaild path.

Overwriting if same database exists (-r or --replace)

If the **-r** option is specified, a new database name overwrites the existing database name if it is identical, insteading outputting an error.

```
cubrid copydb -r -F /home/usr/CUBRID/databases demodb new_demodb
```

Deleting a source database if is is copied (-d or --delete-source)

If the **-d** option is specified, a source database is deleted after the database is copied. This execution brings the same the result as executing **cubrid deletedb** utility after copying a database. Note that if a source database contains LOB data, LOB file directory path of a source database is copied into a new database and it is registered in the **lob-base-path** of the **databases.txt** file.

```
cubrid copydb -d -copyhome/usr/CUBRID/databases demodb new_demodb
```

Copying LOB file directory (--copy-lob-path)

If the **--copy-lob-path** option is specified, a new directory path for LOB files is created and a source database is copied into a new directory path. If this option is omitted, the directory path is not created. Therefore, the **lob-base-path** of the **databases.txt** file should be modified separately. This option cannot be used with the **-B** option.

```
cubrid copydb --copy-lob-path demodb new_demodb
```

Copying LOB file directory simultaneously with specifying it (-B or --delete-backup)

If the **-B** option is specified, a specified directory is specified as for LOB files of a new database and a source database is copied. This option cannot be used with the **--copy-lob-path** option.

```
cubrid copydb -B /home/usr/CUBRID/databases/new_lob demodb new_demodb
```

Installing Database

Description

The **cubrid installdb** utility is used to register the information of a newly installed database to **databases.txt**, which stores database location information. The execution of this utility does not affect the operation of the database to be registered.

Syntax

```
cubrid installdb options database_name
options : [{--server-name=} host] [{-F|--file-path=} database_path] [-L| --log-path=}
log_path]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **installdb** : A command that registers the information of a moved or copied database to **databases.txt**.
- *options* : **--server-name**, **-F**, **-L** options are available. For more information on each option, see the option description and example. If no option is used with a command, the command must be executed in the directory where the corresponding database exists.
- *database_name* : The name of database to be registered to **databases.txt**.

Option

Registering the host name (--server-name)

The following example shows registering the server host information of a database to **databases.txt** with a specific host name. If this option is not specified, the current host information is registered.

```
cubrid installdb --server-name=cub_server1 testdb
```

Registering the directory path of a database volume (-F or ?file-path)

The following example shows registering the directory path of a database volume to **databases.txt** with an **-F** option. If this option is not specified, the path of a current directory is registered as default.

```
cubrid installdb ?F /home/cubrid/CUBRID/databases/testdb testdb
```

Registering the directory path of a database log volume (-L or ?log-path)

The following example shows registering the directory path of a database log volume to **databases.txt** with an **-L** option. If this option is not specified, the directory path of a volume is registered.

```
cubrid installdb ?L /home/cubrid/CUBRID/databases/logs/testdb testdb
```

Checking Used Space

Description

The **cubrid spacedb** utility is used to check how much space of database volumes is being used. It shows a brief description of all permanent data volumes in the database. Information returned by the **cubrid spacedb** utility includes the ID, name, purpose and total/free space of each volume. You can also check the total number of volumes and used/unused database pages.

Syntax

```
cubrid spacedb options database_name
options : [{-o|--output-file=}file] [-S|--SA-mode|-C|--CS-mode] [--size-unit=PAGE|M|G|T|H]
[-s|--summarize]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **spacedb** : A command that checks the space in the database. It executes successfully only when the database is in a stopped state.

- *options* : The **-o**, **-S**, **-C** **--size-unit**, and **-s** options are supported. For details about each option, refer to its description and the examples.
- *database_name* : The name of the database whose space is to be checked. The path-name to the directory where the database is to be created must not be included.

Option

Saving output messages to a file (-o)

The above example shows saving the result of checking the space information of testdb to a file named *db_output*.

```
cubrid spacedb -o db_output testdb
```

Executing in stand-alone mode (-S or --SA-mode)

The **-S** option is used to access a database in standalone, which means it works without processing server; it does not have an argument. If **-S** is not specified, the system recognizes that a database is running in client/server mode.

```
cubrid spacedb --SA-mode testdb
```

Executing in client/server mode (-C or --CS-mode)

The **-C** option is used to access a database in client/server mode, which means it works in client/server process respectively; it does not have an argument. If **-C** is not specified, the system recognize that a database is running in client/server mode by default.

```
cubrid spacedb --CS-mode testdb
```

Outputing in specified size unit (--size-unit)

The **--size-unit** option is used to specify the size unit of the space information of the database to be one of PAGE, M(MB), G(GB), T(TB), H(print-friendly). The default value is **H**. If you set the value to H, the unit is automatically determined as follows: M if 1MB = DB size < 1024MB, G if 1GB = DB size < 1024GB.

```
cubrid spacedb --size_unit=M testdb
cubrid spacedb --size_unit=H testdb
```

Outputs total pages, used pages, free pages by volume usage (-s or --summarize)

Aggregates total_pages, used_pages and free_pages by DATA, INDEX, GENERIC, TEMP and TEMP TEMP, and outputs it.

```
cubrid spacedb –s testdb
```

Compacting Used Space

Description

The **cubrid compactdb** utility is used to secure unused space of the database volume. In case the database server is not running (offline), you can perform the job in stand-alone mode. In case the database server is running, you can perform it in client-server mode.

The **cubrid compactdb** utility secures the space being taken by OIDs of deleted objects and by class changes. When an object is deleted, the space taken by its OID is not immediately freed because there might be other objects that refer to the deleted one. Reference to the object deleted during compacting is displayed as **NULL**, which means this can be reused by OIDs.

Syntax

```
cubrid compactdb options database_name [class_name], class_name2,...]
options : [-v | --verbose] [-S|--SA-mode | -C| --CS-mode]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **compactdb** : A command that compacts the space of the database so that OIDs assigned to deleted data can be reused.

- *options* : The **-v** , **-S**, and **-C** options are supported. Options (**-I**, **-i**, **-c**, **-d**, **-p**) that is applied in client/server mode only.
- *database_name* : The name of the database whose space is to be compacted. The path name to the directory where the database is to be created must not be included.
- *class_name_list* : You can specify the list of tables names that you want to compact space after a database name; the -i option cannot be used together. It is used in client/server mode only.

Option

Outputting detailed messages during execution (-v)

You can output messages that shows which class is currently being compacted and how many instances have been processed for the class by using the **-v** option.

```
cubrid compactdb -v testdb
```

Executing in stand-alone mode (-S or --SA mode)

The **-S** option is specified to compact used space in stand-alone mode while database server is not running; no arugment is specified. If the **-S** option is not specified, system recognizes that the job is executed in client/server mode.

```
cubrid compactdb --SA-mode testdb
```

Executing in client/server mode (C or --CS mode)

The **-C** option is specified to compact used space in client/server mode while database server is running; no argument is specified. Even though this option is omitted, system recognizes that the job is executed in client/server mode. The following options can be used in client/server mode only.

- \- i, --input-class-file=FILE

  You can specify an input file name that contains the table table name with this option. Write one table name in a single line; invalid table name is ignored. Note that you cannot specify the list of the table names after a database name in case of you use this option.
- -p, --page-commited-once = NUMBER

  You can specify the number of maximum pages that can be commited once with this option. The default value is 10, the minimum value is 1, and the maximum value is 10. The less option value is specified, the more concurrency is enhanced because the value for class/instance lock is small; however, it causes slowdown on operation, and vice versa.
- -d, --delete-old-repr

  You can delete an existing table representation from catalog with this option.
- -I, --Instance-lock-timeout

  You can specify a value of instance lock timeout with this option. The default value is 2(second), the minimum value is 1, and the maximum value is 10. The less option value is specified, the more operation speeds up. However, the number of instances that can be processed becomes smaller, and vice versa.
- -c, --class-lock-timeout

  You can specify a value of instance lock timeout with this option. The default value is 10(second), the minimum value is 1, and the maximum value is 10. The less option value is specified, the more operation speeds up. However, the number of tables that can be processed becomes smaller, and vice versa.

```
cubrid compactdb --CS-mode -p 10 testdb tbl1, tbl2, tbl5
```

Updating Statistics

Description

Updates statistical information such as the number of objects, the number of pages to access, and the distribution of attribute values.

Syntax

```
cubrid optimizedb options database_name
options : [{-n|--class-name=} name]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **optimizedb** : Updates the statistics information, which is used for cost-based query optimization of the database. If the option is specified, only the information of the specified class is updated.
- *options* : The **-n** option is supported.
- *database_name* : The name of the database whose cost-based query optimization statistics are to be updated.

Option

Updating the query statistics of the target database

The following example shows updating the query statistics information of all classes in the database.

```
cubrid optimizedb testdb
```

Updating the query statistics of a specific class in the database (-n or --class-name)

The following example shows updating the query statistics information of the given class by using the **-n** option.

```
cubrid optimizedb -n event_table testdb
```

Outputting Statistics Information of Server

Description

The cubrid statdump utility checks statistics information processed by the CUBRID database server. The statistics information mainly consists of the followings: File I/O, Page buffer, Logs, Transactions, Concurrency/Lock, Index, and Network request

Note that you must specify the parameter **communication_histogram** to **yes** in the **cubrid.conf** before executing the utility. You can also check statistics information of server with session commands (**;.h on**) in the CSQL.

Syntax

```
cubrid statdump options database_name
options :
[{-o |--ouput-file=}file_name] [{-i |--interval=}secs] [-c|--cumulative] [{-s |-
substr=}sub_string]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **installdb** : A command that dumps the statistics information on the database server execution.
- *options* : **--o**, **-i**, **-c**, and **-s** options are available.
- *database_name* : The name of database which has the statistics data to be dumped.

Option

Outputting statistics information periodically (-i or --interval)

```
cubrid statdump -i 5 testdb

Thu April 07 23:10:08 KST 2011

 *** SERVER EXECUTION STATISTICS ***
Num_file_creates              =          0
Num_file_removes              =          0
Num_file_ioreads              =          0
Num_file_iowrites             =          0
Num_file_iosynches            =          0
Num_data_page_fetches         =          0
Num_data_page_dirties         =          0
Num_data_page_ioreads         =          0
Num_data_page_iowrites        =          0
```

```
Num_data_page_victims         =          0
Num_data_page_iowrites_for_replacement =          0
Num_log_page_ioreads          =          0
Num_log_page_iowrites         =          0
Num_log_append_records        =          0
Num_log_archives              =          0
Num_log_checkpoints           =          0
Num_log_wals                  =          0
Num_page_locks_acquired       =          0
Num_object_locks_acquired     =          0
Num_page_locks_converted      =          0
Num_object_locks_converted    =          0
Num_page_locks_re-requested   =          0
Num_object_locks_re-requested =          0
Num_page_locks_waits          =          0
Num_object_locks_waits        =          0
Num_tran_commits              =          0
Num_tran_rollbacks            =          0
Num_tran_savepoints           =          0
Num_tran_start_topops         =          0
Num_tran_end_topops           =          0
Num_tran_interrupts           =          0
Num_btree_inserts             =          0
Num_btree_deletes             =          0
Num_btree_updates             =          0
Num_btree_covered             =          0
Num_btree_noncovered          =          0
Num_btree_resumes             =          0
Num_query_selects             =          0
Num_query_inserts             =          0
Num_query_deletes             =          0
Num_query_updates             =          0
Num_query_sscans              =          0
Num_query_iscans              =          0
Num_query_lscans              =          0
Num_query_setscans            =          0
Num_query_methscans           =          0
Num_query_nljoins             =          0
Num_query_mjoins              =          0
Num_query_objfetches          =          0
Num_network_requests          =          1
Num_adaptive_flush_pages      =          0
Num_adaptive_flush_log_pages  =          0
Num_adaptive_flush_max_pages  =        900

 *** OTHER STATISTICS ***
Data_page_buffer_hit_ratio    =       0.00
```

Category of Statistics Information

Category	**Item**	**Description**
File I/O	Num_file_removes	The number of files removed
	Num_file_creates	The number of files created
	Num_file_ioreads	The number of files read
	Num_file_iowrites	The number of files saved
	Num_file_iosynches	The number of file synchronization
Page buffer	Num_data_page_fetches	The number of pages fetched
	Num_data_page_dirties	The number of duty pages
	Num_data_page_ioreads	The number of pages read
	Num_data_page_iowrites	The number of pages saved
	Num_data_page_victims	The number specifying the victim data to be flushed from the data page to the disk

	Num_data_page_iowrites_for_replacement	The number of the written data pages specified as victim
	Num_adaptive_flush_pages	The number of data pages flushed from the data buffer to the disk
	Num_adaptive_flush_log_pages	The number of log pages flushed from the log buffer to the disk
	Num_adaptive_flush_max_pages	The maximum number of pages allowed to flush from data and the log buffer to the disk
Logs	Num_log_page_ioreads	The number of log pages read
	Num_log_page_iowrites	The number of log pages saved
	Num_log_append_records	The number of log records appended
	Num_log_archives	The number of logs archived
	Num_log_checkpoints	The number of checkpoints
	Num_log_wals	Not used
Transactions	Num_tran_commits	The number of commits
	Num_tran_rollbacks	The number of rollbacks
	Num_tran_savepoints	The number of savepoints
	Num_tran_start_topops	The number of top operations started
	Num_tran_end_topops	The number of top perations stopped
	Num_tran_interrupts	The number of interruptions
Concurrency/lock	Num_page_locks_acquired	The number of locked pages acquired
	Num_object_locks_acquired	The number of locked objects acquired
	Num_page_locks_converted	The number of locked pages converted
	Num_object_locks_converted	The number of locked objects converted
	Num_page_locks_re-requested	The number of locked pages requested
	Num_object_locks_re-requested	The number of locked objects requested
	Num_page_locks_waits	The number of locked pages waited
	Num_object_locks_waits	The number of locked objects waited
Index	Num_btree_inserts	The number of nodes inserted
	Num_btree_deletes	The number of nodes deleted
	Num_btree_updates	The number of nodes updated
	Num_btree_covered	The number of cases in which an index includes all data upon query execution

	Num_btree_noncovered	The number of cases in which an index includes some or no data upon query execution
	Num_btree_resumes	The exceeding number of index scan specified in index_scan_oid_buffer_pages
Query (Servuce Workload)	Num_query_selects	The number of SELECT requested
	Num_query_inserts	The number of INSERT queries
	Num_query_deletes	The number of DELETE queries
	Num_query_updates	The number of UPDATE queries
	Num_query_sscans	The number of sequential scans (full scan)
	Num_query_iscans	The number of index scans
	Num_query_lscans	The number of LIST scans
	Num_query_setscans	The number of SET scans
	Num_query_methscans	The number of METHOD scans
	Num_query_nljoins	The number of nested loop joins
	Num_query_mjoins	The number of parallel joins
	Num_query_objfetches	The number of fetch objects
Network request	Num_network_requests	The number of networks requested
	Data_page_buffer_hit_ratio	Hit Ratio of page buffers (Num_data_page_fetches - Num_data_page_ioreads)*100 / Num_data_page_fetches

Saving statistics information to a file (-o or --output-file)

The **-o** options is used to save statistics information of server processing for the database to a specified file.

```
cubrid statdump -o statdump.log testdb
```

Displays the accumulated operation statistics information (-c or --cumulative)

You can display the accumulated operation statistics information of the target database server by using the **-c** option. By combining this with the ?i option, you can check the operation statistics information at a specified interval.

```
cubrid statdump ?i 5 ?c testdb
```

Displays statistics that includes specified string (-s or --substr)

You can display statistics about items of which name include the specified string by using **-s** option.

The following examples displays statistics about items of which name include "data".

```
cubrid statdump –s data testdb

*** SERVER EXECUTION STATISTICS ***
Num_data_page_fetches         =        135
Num_data_page_dirties         =          0
Num_data_page_ioreads         =          0
Num_data_page_iowrites        =          0
Num_data_page_victims         =          0
Num_data_page_iowrites_for_replacement =          0

 *** OTHER STATISTICS ***
Data_page_buffer_hit_ratio    =     100.00
```

Note Each status information consists of 64-bit INTEGER data and the corresponding statistics information can be lost if the accumulated value exceeds the limit.

Checking Lock Status

Description

The **cubrid lockdb** utility is used to check the information on the lock being used by the current transaction in the database.

Syntax

```
cubrid lockdb options database_name
options : [{-o|--output-file=} file ]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **lockdb** : A command used to check the information on the lock being used by the current transaction in the database.
- *options* : The **-o** option is supported.
- *database_name* : The name of the database where lock information of the current transaction is to be checked.

Option

Displaying the lock information on the screen

The following example shows displaying lock information of the testdb database on a screen without any option.

```
cubrid lockdb testdb
```

Displaying the lock information to the specified file (-o)

The following example shows displaying lock information of the testdb database as a output.txt by using the **-o** option.

```
cubrid lockdb -o output.txt testdb
```

Checking Database Consistency

Description

The **cubrid checkdb** utility is used to check the consistency of a database. You can use **cubrid checkdb** to identify data structures that are different from indexes by checking the internal physical consistency of the data and log volumes. If the **cubrid checkdb** utility reveals any inconsistencies, you must try automatic repair by using the **--repair** option.

Syntax

```
cubrid checkdb options database_name [class_name1 class_name2 ...]
options : [-S|--SA-mode | -C|--CS-mode] [-r | --repair] | [-i table_list.txt|--input-
class-file]
```

- **cubrid** : An integrated utility for CUBRID service and database management.
- **checkdb** : A utility that checks the data consistency of a specific database.
- *options* : **-S**, **-C**, **-r**, and **-i** options are supported.
- *database_name* : The name of the database whose consistency status will be either checked or repaired.

```
table_list.txt : A file name to save the list of the tables for consistency check or
recovery
class_name1 class_name2 : List the table names for consistency check or recovery
```

Option

Checking the database consistency in standalone mode (-S or --SA-mode)

The **-S** option is used to access a database in standalone, which means it works without processing server; it does not have an argument. If **-S** is not specified, the system recognizes that a database is running in client/server mode.

```
cubrid checkdb -S testdb
```

Checking the database consistency in client/server mode (-C or --CS-mode)

The **-C** option is used to access a database in client/server mode, which means it works in client/server process respectively; it does not have an argument. If **-C** is not specified, the system recognize that a database is running in client/server mode by default.

```
cubrid checkdb -C testdb
```

Repairing in case of a database consistency problem (-r or --repair)

The **-r** option is used to repair an issue if a consistency error occurs in a database.

```
cubrid checkdb -r testdb
```

Limit to the tables specifying the target for consistency check or recovery of database (-i, --input-class-file or tables)

You can limit the target for consistency check or recovery as in the table list file after **-i** *table_list.txt* option or the tables specified after the database name. They can be used together and if the target is not specified, full database consistency check or recovery will be executed.

```
cubrid checkdb testdb tbl1 tbl2
cubrid checkdb -r testdb tbl1 tbl2
cubrid checkdb -r -i tbl_list.txt testdb tbl1 tbl2
```

Empty string, tab, carriage return and comma are separators among table names in the table list file specified by **-i** option. The following is an example of a table list file and recognizes all from t1 to t10 as tables for consistency check or recovery.

```
t1 t2 t3,t4 t5
t6, t7 t8   t9

     t10
```

Killing Database Transactions

Description

The **cubrid killtran** is used to check transactions or abort specific transaction. Only a DBA can execute this utility.

Syntax

```
cubrid killtran options database_name
options :
[{-i|--kill-transaction-index=}index] [--kill-user-name=id] [--kill-host-name=host] [--
kill-program-name=program_name] [{-p|--dba-password=}password] [-d|--display-information]
[-f|--force]
```

- **cubrid** : An integrated utility for the CUBRID service and database management
- **killtran** : A utility that manages transactions for a specified database
- *options* : Some options refer to killing specified transactions; others refer to outputting active transactions. If no option is specified, **-d** is specified by default so all transactions are outputted on the screen. **-p** A value followed by the -p option is a password of the **DBA**, and should be entered in the prompt.
- *database_name* : The name of database whose transactions are to be killed

Option

Outputting all transactions (no option)

```
cubrid killtran testdb
```

```
Tran index      User name      Host name      Process id      Program name
-------------------------------------------------------------------------------
      1(+)            dba      myhost              664           cub_cas
      2(+)            dba      myhost             6700              csql
      3(+)            dba      myhost             2188           cub_cas
      4(+)            dba      myhost              696              csql
      5(+)         public      myhost             6944              csql
-------------------------------------------------------------------------------
```

Killing transactions in a specified index (-i or --kill-transation-index)

```
cubrid killtran -i 1 testdb

Ready to kill the following transactions:

Tran index      User name      Host name      Process id      Program name
-------------------------------------------------------------------------------
      1(+)            dba      myhost             4760              csql
-------------------------------------------------------------------------------
Do you wish to proceed ? (Y/N)y
Killing transaction associated with transaction index 1
```

Outputting all transactions (-d or --display)

```
cubrid killtran -d testdb

Tran index      User name      Host name      Process id      Program name
-------------------------------------------------------------------------------
      2(+)            dba      myhost             6700              csql
      3(+)            dba      myhost             2188           cub_cas
      4(+)            dba      myhost              696              csql
      5(+)         public      myhost             6944              csql
-------------------------------------------------------------------------------
```

Killing transactions for a specified OS user ID (--kill-user-name)

```
cubrid killtran --kill-user-name=os_user_id testdb
```

Killing transactions for a specified client host (--kill- host-name)

```
cubrid killtran --kill-host-name=myhost testdb
```

Killing transactions for a specified program (--kill-program-name)

```
cubrid killtran --kill-program-name=cub_cas testdb
```

Omitting a prompt to check transactions to be stopped (-f or --force)

```
cubrid killtran -f -i 1 testdb
```

Checking the Query Plan Cache

Description

The **cubrid plandump** utility is used to display information on the query plans saved (cached) on the server.

Syntax

```
cubrid plandump options database_name
options : [-d|--drop]  [{-o|--output-file=} file]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **plandump** : A utility that displays the query plans saved in the current cache of a specific database.
- *options* : The **-d** and **-o** options are supported.
- *database_name* : The name of the database where the query plans are to be checked or dropped from its sever cache.

Option

Checking the query plans saved in the cache

```
cubrid plandump testdb
```

Dropping the query plans saved in the cache (-d or --drop)

```
cubrid plandump -d testdb
```

Saving the results of the query plans saved in the cache to a file (-o or --output)

```
cubrid plandump -o output.txt testdb
```

Outputting Internal Database Information

Description

You can check various pieces of internal information on the database with the **cubrid diagdb** utility. Information provided by **cubrid diagdb** is helpful in diagnosing the current status of the database or figuring out a problem.

Syntax

```
cubrid diagdb options database_name
options : [{-d | --dump-type=} type]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **diagdb** : A command that is used to check the current storage state of the database by outputting the information contained in the binary file managed by CUBRID in text format. It normally executes only when the database is in a stopped state. You can check the whole database or the file table, file size, heap size, class name or disk bitmap selectively by using the provided option.
- *options* : The **-d** option is provided.
- *database_name* : The name of the database to be diagnosed.

Option

Specifying the output range (-d or --dump-type)

The following example displays the information of all files in the testdb database. If any option is not specified, the default value of 1 is used.

```
cubrid diagdb -d 1 myhost testdb
```

The utility has 9 types of -d options as follows:

Type	Description
-1	Outputs all database information.
1	Outputs file table information.
2	Outputs file capacity information.
3	Outputs heap capacity information.
4	Outputs index capacity information.
5	Outputs class name information.
6	Outputs disk bitmap information.
7	Outputs catalog information.
8	Outputs log information.
9	Outputs hip information.

Backup and Restore

DBA must perform regular backups of the database so that it can be restored successfully to a state at a certain point in time in case of system failure. For more information, see Database Backup.

Export and Import

To use a newer version of CUBRID database, the existing version must be migrated to a new one. For this purpose, you can use "Export to a ASCII text file" and "Import from a ASCII text file" features provided by CUBRID. For more information on export and import, see Migrating Database.

Outputting Parameters Used in Server/Client

Description

The **cubrid paramdump** utility outputs parameter information used in the server/client process.

Syntax

```
cubrid paramdump options database_name
options : [{-o|--output-file=}filename]  [{-b|--both}]  [{-S|--SA-mode}]  [{-C|--CS-mode}]
```

- **cubrid** : An integrated utility for the CUBRID service and database management
- **paramdump** : A utility that outputs parameter information used in the server/client process
- *options* : A short name option starts with a single dash (-) while a full name option starts with a double dash (--). **-o**, **-b**, **-S** and **-C** options are provided.
- *database_name* : The name of the database in which parameter information is to be outputted

Option

Saving the output information to a file (-o)

The **-o** option is used to save information of the parameters used in the server/client process of the database into a specified file. The file is created in the current directory. If the **-o** option is not specified, messages are displayed on the console screen.

```
cubrid paramdump -o db_output testdb
```

Outputting information of the server/client parameters (-b)

The **-b** option is used to output parameter information used in server/client process into a console screen. If the **-b** option is not specified, only server-side information is outputted.

```
cubrid paramdump -b testdb
```

Outputting parameter information of the server process in standalone mode (-S or --SA-mode)

```
cubrid paramdump -S testdb
```

Outputting parameter information of the server process in client/server mode (-C or --CS-mode)

```
cubrid paramdump -C testdb
```

Database Migration

Migrating Database

To use a newer version of CUBRID database, you might migrate an existing data to a new one. For this purpose, you can use the "Export to a ASCII text file" and "Import from a ASCII text file" features provided by CUBRID. The following section explains migration steps using the **cubrid unloaddb** and **cubrid loaddb** utilities.

Recommended scenario and procedures

The following is an explanation of a migration scenario that can be applied while the existing version of CUBRID is running. For database migration, the **cubrid unloaddb** and **cubrid loaddb** utilities are used. For more information, see Unloading Database and Loading Database.

1. Back up the existing database

   Back up the existing version of the database by using the **cubrid backupdb** utility. The purpose of this step is to safeguard against failures that might occur during the database unload/load operations. For more information on the database backup, see Database Backup.

2. Unload the existing database

   Unload the database created for the existing version of CUBRID by using the **cubrid unloaddb** utility. For more information on the database unload, see Unloading Database.

3. Storing the existing CUBRIDG configuration files

   Save configurations files such as **cubrid.conf**, **cubrid_broker.conf** and **cm.conf** located in the **CUBRID/conf** directory. The purpose of this step is to conveniently apply parameter values for the existing CUBRID database environment to the new one.

4. Install a new version of CUBRID

   Once backing up and unloading of the data created by the existing version of CUBRID have been completed, delete the existing version of CUBRID and its databases and then install the new version of CUBRID. For more information on installing CUBRID, see Installing and Running on Linux in "Getting Started."

5. Configure the new CUBRID

   You can configure the new version of CUBRID by referring to configuration files of the existing database saved in the step 3, "**Save configuration files of the existing version of CUBRID**." For more information on configuration, see Installing and Running on Windows in "Getting Started."

6. Load the new database

   Create a database by using the **cubrid createdb** utility and then use the **cubrid loaddb** utility to load into the new database the data which had previously been unloaded. For more information on creating a database, see Creating Database in "Administrator's Guide." For more information on database loading, see Loading Database.

7. Back up the new database

   Once the data has been successfully loaded into the new database, back up the database created for the new version of CUBRID by using the **cubrid backupdb** utility. The reason for this step is because you cannot restore the data backed up in the existing version of CUBRID when using the new version. For more information on backing up the database, see Database Backup.

Unloading Database

Description

The purposes of unloading/loading a database are as follows:

- To reconstruct the database by rebuilding the database volume
- To perform migration to a different system environment
- To perform migration to a different version of the DBMS

Syntax

```
cubrid unloaddb [ options ] database_name
[ options ]
-i | -O | -s | -d | -v | -S | -C |
--input-class-file | --output-path | --schema-only | --data-only | --verbose | --SA-mode |
--CS-mode | --include-reference | --input-class-only | --lo-count | --estimated-size | --
cached-pages | --output-prefix | --hash-file | --datafile-per-class
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **unloaddb** : A utility that creates ASCII files from a database. It is used together with the **cubrid loaddb** utility for replacing system, upgrading product version or reorganizing database volumes. It can be used both in standalone and client/server modes. Data can be unloaded even when the database is running.
- *options* : A short option starts with a single dash (-) while a full name option starts with a double dash (--). Note that options are case sensitive.
- *database_name* : Specifies the name of the database to be unloaded.

Return value

Return values of **cubrid unloaddb** utility are as follows:

- 0 : Success
- Non-zero : Failure

Generated Files

- Schema file (*database-name*_**schema**) : A file that contains information on the schema defined in the database.
- Object file (*database-name*_**objects**) : A file that contains information on the records in the database.
- Index file (*database-name*_**indexes**) : A file that contains information on the indexes defined in the database.
- Trigger file (*database-name*_**trigger**) : A file that contains information on the triggers defined in the database. If you don't want triggers to be running while loading the data, load the trigger definitions after the data loading has completed.

Schema, object, index and trigger files are created in the same directory.

Option

The following table shows options that can be used with **cubrid unloaddb** utility. Options are case sensitive.

Option	Description
-i --input-class-file	Unloads the database class into the input file specified in an argument.
-O --output-path	Specifies the directory in which to create schema and object files. If the option is not specified, files are created in the current directory.
-s --schema-only	Creates only the schema file, not the data file.
-d --data-only	Creates only the data file, not the schema file.
-v --verbose	Displays detailed information on the database being unloaded.
-S --SA-mode	Unloads the database in standalone mode.
-C --CS-mode	Unloads the database in client/server mode.
--include-reference	Unloads the object reference as well when the specified database class is unloaded with the **-i** option.
--input-class-only	Is used with the **-i** option. Creates only the schema files which are related to tables included in the input file.

--lo-count	Specifies the number of large object (LO) data files to be created in a single directory. Default value : 0
--estimated-size	Specifies the number of records expected.
--cached-pages	Configures the number of object tables to be cached in the memory. Default value : 100
--output-prefix	Specifies the prefix for schema and object file names.
--hash-file	Specifies the name of the hash file.
--datafile-per-class	Generates a data file per each table.

Input file with the list of tables to be unloaded (-i or --input-class-file)

The following is an example of a input file (table_list.txt).

```
table_1
table_2
..
table_n
```

The **-i** option specifies the input file where the list of tables to be unloaded is stored so that only specified part of the database can be unloaded.

```
cubrid unloaddb -i table_list.txt demodb
```

The **-i** option can be used together with the **--input-class-only** option that creates the schema file related to only those tables included in the input file.

```
cubrid unloaddb --input-class-only -i table_list.txt demodb
```

The **-i** option can be used together with the **--include-reference** option that creates the object reference as well.

```
cubrid unloaddb --include-reference -i table_list.txt demodb
```

Specifying the directory where files created will be saved (-O or --output-path)

The **-O** option specifies the directory where the output files generated by the unload operation is saved. If the **-O** option is not specified, output files are created in the current working directory.

```
cubrid unloaddb -O ./CUBRID/Databases/demodb demodb
```

If the specified directory does not exist, the following error message will be displayed.

```
unloaddb: No such file or directory.
```

Creating the schema file only (-s or --schema-only)

The **-s** option specifies that only the schema file will be created from amongst all the output files which can be created by the unload operation.

```
cubrid unloaddb -s demodb
```

Creating the data file only (-d or -data-only)

The **-d** option specifies that only the data file will be created from amongst all of the output files which can be created by the unload operation.

```
cubrid unloaddb -d demodb
```

Creates data files by table (--datafile-per-class)

--datafile-per-class is the option specifying that the output file generated through unload operation creates a data file per each table. The file name is generated as *<Database Name>*_*<Table Name>***_objects** for each table. However, all column values in object types are unloaded as NULL and %id class_name class_id part is not written in the unloaded file (see How to Write a File to Load Database).

```
cubrid unloaddb -d demodb
```

Displaying the unload status information (-v or --verbose)

The **-v** option displays detailed information on the database tables and records being unloaded while the unload operation is under way.

```
cubrid unloaddb -v demodb
```

Standalone mode (-S or --SA-mode)

The **-S** option performs the unload operation by accessing the database in standalone mode.

```
cubrid unloaddb -S demodb
```

Client/server mode (-C or --CS-mode)

The **-C** option performs the unload operation by accessing the database in client/server mode.

```
cubrid unloaddb -C demodb
```

Number of estimated records (--estimated-size)

The **--estimated-size** option allows you to assign hash memory to save records of the database to be unloaded. If the **--estimated-size** option is not specified, the number of records of the database is determined based on recent statistics information. This option can be used if the recent statistics information has not been updated or if a large amount of hash memory needs to be assigned. Therefore, if the number given as the argument for the option is too small, the unload performance deteriorates due to hash conflicts.

```
cubrid unloaddb --estimated-size 1000 demodb
```

Number of pages to be cached (--cached-pages)

The **--cached-pages** option specifies the number of pages of tables to be cached in the memory. Each page is 4,096 bytes. The administrator can configure the number of pages taking into account the memory size and speed. If this option is not specified, the default value is 100 pages.

```
cubrid unloaddb --cached-pages 500 demodb
```

Specifying the prefix for the name of the file to be created (--output-prefix)

The **--output-prefix** option specifies the prefix for the names of schema and object files created by the unload operation. Once the example is executed, the schema file name becomes abcd_schema and the object file name becomes abcd_objects. If the **--output-prefix** option is not specified, the name of the database to be unloaded is used as the prefix.

```
cubrid unloaddb --output-prefix abcd demodb
```

Loading Database

Description

You can load a database by using the **cubrid loaddb** utility in the following scenarios:

- When migrating a previous CUBRID database version to a new version
- When migrating a database of third-party DBMS to a CUBRID database
- When entering mass data faster than executing the **INSERT** statement

Generally, the **cubrid loaddb** utility uses files created by the **cubrid unloaddb** utility (schema definition file, object input file and index definition file).

Syntax

```
cubrid loaddb [ options ] database_name
[ options ]
-u | -p | -l | -v | -c | -s | -i | -d |
--user | --password | --load-only | --verbose | --periodic-commit |  --schema-file | --
index-file | --data-file | --data-file-check-only | --estimated-size | --no-oid | --no-
statistics | --ignore-class-file |--error-control-file |
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **loaddb** : A utility loads files which is generated by the unload operation and then creates a new database. It is also used to enter mass data into a database faster than ever by loading the input file written by a user. Database loading is performed in standalone mode with **DBA** authorization.
- *options* : A short name option starts with a single dash (-) while a full name option starts with a double dash (--). The options are case sensitive.
- *database_name* : Specifies the name of the database to be created.

Return value

Return values of **cubrid loaddb** utility are as follows:

- 0 : Success
- Non-zero : Failure

Input file

- Schema file (*database-name*_**schema**) : A file generated by the unload operation; it contains schema information defined in the database.
- Object file (*database-name*_**objects**) : A file created by an unload operation. It contains information on the records in the database.
- Index file (*database-name*_**indexes**) : A file created by an unload operation. It contains information on the indexes defined in the database.
- Trigger file (*database-name*_**trigger**) : A file created by an unload operation. It contains information on the triggers defined in the database.
- User-defined object file (*user_defined_object_file*) : A file in table format written by the user to enter mass data.

Option

The following table shows options that can be used with **cubrid loaddb** utility. The options are case sensitive.

Option	Description
-u **--user**	Enters the database user's account. The default value is **PUBLIC**.
-p **--password**	Enters the database user's password.
-l **--load-only**	Skips checking statements and types included in the object file and loads records.
-v **--verbose**	Displays detailed information on the data loading status on the screen.
-c **--periodic-commit**	Commits the transaction whenever a specified number of records has been entered.
-s **--schema-file**	Specifies the schema file created by the unload operation and performs schema loading.
-i **--index-file**	Specifies the index file created by the unload operation and loads indexes.
-d **--data-file**	Specifies the data file created by the unload operation and loads records.
--data-file-check-only	Performs checking only for statements and types included in the data file, but does not load records.
--estimated-size	Specifies the number of records expected.
--no-oid	Ignores the OID reference relationship included in the data file and loads records.
--no-statistics	Loads records without updating database statistics information.
--ignore-class-file	Specifies the ignoring classes.

--error-control-file	Specifies the file that describes how to handle specific errors occurring during data loading.

Entering a user account (-u or --user)

The **-u** option specifies the user account of a database where records are loaded. If the option is not specified, the default value is **PUBLIC**.

```
cubrid loaddb -u admin -d demodb_objects newdb
```

Entering the password (-p or --password)

The **-p** option specifies the password of a database user who will load records. If the option is not specified, you will be prompted to enter the password.

```
cubrid loaddb -p admin -d demodb_objects newdb
```

Loading records without checking syntax (-l or --load-only)

The **-l** option loads data directly without checking the syntax for the data to be loaded. The following example is a statement that loads data included in demodb_objects to newdb.
If the **-l** option is used, loading speed increases because data is loaded without checking the syntax included in demodb_objects, but an error might occur.

```
cubrid loaddb -l -d demodb_objects newdb
```

Displaying the loading status information (-v or --verbose)

The following is a statement that outputs detailed information on the tables and records of the database being loaded while the database loading operation is performed. You can check the detailed information such as the progress level, the class being loaded and the number of records entered by using the **-v** option.

```
cubrid loaddb -v -d demodb_objects newdb
```

Configuring the commit interval (-c or --periodic-commit)

The following command performs commit regularly every time 100 records are entered into the newdb by using the **-c** option. If the **-c** option is not specified, all records included in demodb_objects are loaded to newdb before the transaction is committed. If the **-c** option is used together with the **-s** or **-i** option, commit is performed regularly every time 100 DDL statements are loaded. The recommended commit interval varies depending on the data to be loaded. It is recommended that the parameter of the **-c** option be configured to 50 for schema loading, 1,000 for record loading, and 1 for index loading.

```
cubrid loaddb -c 100 -d demodb_objects newdb
```

Schema loading (-s or --schema-file)

The following statement loads the schema information defined in demodb into the newly created newdb database. demodb_schema is a file created by the unload operation and contains the schema information of the unloaded database. You can load the actual records after loading the schema information first by using the **-s** option.

```
cubrid loaddb -u dba -s demodb_schema newdb

Start schema loading.
Total 86 statements executed.
Schema loading from demodb_schema finished.
Statistics for Catalog classes have been updated.
```

Index loading (-i or --index-file)

The following command loads the index information defined in demodb into the newly created newdb database. demo_indexes is a file created by the unload operation and contains the index information of the unloaded database. You can create indexes after loading records by using the **-i** option together with the **-d** option.

```
cubrid loaddb -u dba -i demodb_indexes newdb
```

Data loading (-d or -data-file)

The following command loads the record information into newdb by specifying the data file or the user-defined object file with the **-d** option. demodb_objects is either an object file created by the unload operation or a user-defined object file written by the user for mass data loading.

```
cubrid loaddb -u dba -d demodb_objects newdb
```

Checking the syntax for the data to be loaded only (--data-file-check-only)

The following is a command that checks the statements for the data contained in demodb_objects by using the **--data-file-check-only** option. Therefore, the execution of the command below does not load records.

```
cubrid loaddb --data-file-check-only -d demodb_objects newdb
```

Number of expected records (--estimated-size)

The **--estimated-size** option can be used to improve loading performance when the number of records to be unloaded exceeds the default value of 5,000. That is, you can improve the load performance by assigning large hash memory for record storage with this option.

```
cubrid loaddb --estimated-size 8000 -d demodb_objects newdb
```

Loading records while ignoring the reference relationship (--no-oid**)**

The following is a command that loads records into newdb ignoring the OIDs in demodb_objects.

```
cubrid loaddb --no-oid -d demodb_objects newdb
```

Loading records without updating statistics information (--no-statistics)

The following is a command that does not update the statistics information of newdb after loading demodb_objects. It is useful especially when small data is loaded to a relatively big database; you can improve the load performance by using this command.

```
cubrid loaddb --no-statistics -d demodb_objects newdb
```

Specifying the ignoring classes (--ignore-class-file**)**

You can specify a file that lists classes to be ignored during loading records. All records of classes except ones specified in the file will be loaded.

```
cubrid loaddb --ignore-class-file=skip_class_list -d demodb_objects newdb
```

Specifying the error information file (--error-control-file)

This option specifies the file describing how to handle specific errors occurring during database loading.

```
cubrid loaddb --error-control-file=error_test -d demodb_objects newdb
```

How to Write Files to Load Database

You can add mass data to the database more rapidly by writing the object input file used in the **cubrid loaddb** utility. An object input file is a text file in simple table form that consists of comments and command/data lines.

Comment

In CUBRID, a comment is represented by two hyphens (--).

```
-- This is a comment!
```

Command Line

A command line begins with a percent character (%) and consists of **%class** and **%id** commands; the former defines classes, and the latter defines aliases and identifiers used for class identification.

Assigning an identifier to a class

You can assign an identifier to class reference relationships by using the **%id** command.

Syntax

```
%id class_name class_id
class_name:
    identifier
class_id:
    integer
```

The *class_name* specified by the **%id** command is the class name defined in the database, and *class_id* is the numeric identifier which is assigned for object reference.

Example 1

```
%id employee 2
%id office 22
%id project 23
%id phone 24
```

Specifying the class and attribute

You can specify the classes (tables) and attributes (columns) upon loading data by using the **%class** command. The data line should be written based on the order of attributes specified.

Syntax

```
%class class_name ( attr_name [ { attr_name }_ ]
```

The schema must be pre-defined in the database to be loaded.

The *class_name* specified by the **%class** command is the class name defined in the database and the *attr_name* is the name of the attribute defined.

Example 2

The following is an example that specifies a class and three attributes by using the **%class** command to enter data into a class named employee. Three pieces of data should be entered on the data lines after the **%class** command. For this, see Example 3 in the "Configuring a reference relationship" section.

```
%class employee (name age department)
```

Data Line

A data line comes after the **%class** command line. Data loaded must have the same type as the class attributes specified by the **%class** command. The data loading operation stops if these two types are different.

Data for each attribute must be separated by at least one space and be basically written as a single line. However, if the data to be loaded takes more than one line, you should specify the plus sign (+) at the end of the first data line to enter data continuously on the following line. Note that no space is allowed between the last character of the data and the plus sign.

Loading an instance

As shown below, you can load an instance that has the same type as the specified class attribute. Each piece of data is separated by at least one space.

Example 1

```
%class employee (name)
'jordan'
'james'
'garnett'
'malone'
```

Assigning an instance number

You can assign a number to a given instance at the beginning of the data line. An instance number is a unique positive number in the specified class. Spaces are not allowed between the number and the colon (:). Assigning an instance number is used to configure the reference relationship for later.

Example 2

```
%class employee (name)
1: 'jordan'
2: 'james'
3: 'garnett'
4: 'malone'
```

Configuring a reference relationship

You can configure the object reference relationship by specifying the reference class after an "at sign (**@**)" and the instance number after the "vertical line (|)."

Syntax

```
@class_ref | instance_no
class_ref:
     class_name
     class_id
```

Specify a class name or a class id after the **@** sign, and an instance number after a vertical line (|). Spaces are not allowed before and after a vertical line (|).

Example 3

The following is an example that loads class instances into the paycheck class. The name attribute references an instance of the employee class. As in the last line, data is loaded as **NULL** if you configure the reference relationship by using an instance number not specified earlier.

```
%class paycheck(name department salary)
@employee|1   'planning'   8000000
@employee|2   'planning'   6000000
@employee|3   'sales'   5000000
@employee|4   'development'   4000000
@employee|5   'development'   5000000
```

Example 4

Since the id 21 was assigned to the employee class by using the **%id** command in the Assigning an identifier to a class section, Example 3 can be written as follows:

```
%class paycheck(name department salary)
@21|1   'planning'   8000000
@21|2   'planning'   6000000
@21|3   'sales'   5000000
@21|4   'development'   4000000
@21|5   'development'   5000000
```

Database Backup and Restore

Database Backup

A database backup is the procedure of storing CUBRID database volumes, control files and log files, and it is executed by using the **cubrid backupdb** utility or the CUBRID Manager. **DBA** must regularly back up the database so that the database can be properly restored in the case of storage media or file errors. The restore environment must have the same operating system and the same version of CUBRID as the backup environment. For such a reason, you must perform a backup in a new environment immediately after migrating a database to a new version.

To recover all database pages, control files and the database to the state at the time of backup, the **cubrid backupdb** utility copies all necessary log records.

Syntax

```
cubrid backupdb [ options ] database_name
[ options ]
-D | -r | -l | -o | -S | -C | -t | -z | -e |
--destination-path | --remove-archive | --level | --output-file | --SA-mode | --CS-mode |
--thread-count | --compress | --except-active-log | --no-check
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **backupdb** : A utility for database backup. Performs an online, offline, compressed or parallel backup depending on the option used. This utility can only be executed by a user who has the backup authorization (e.g. **DBA**).
- *options* : A short option starts with a single dash (-) while a full name option starts with a double dash (--). Options are case sensitive.
- *database_name* : Specifies the name of the database to be backed up.

Return Value

- 0 : Success
- Non-zero : Failure

Option

The following table shows options that can be used with **cubrid backupdb** utility. Note that options are case sensitive.

Option	Description
-D --destination-path	Specifies the directory path name or device name where backup volumes are to be created. The default value is the location of **log_path** specified in the database location file (**databases.txt**) which was generated upon database creation.
-r --remove-archive	Removes unnecessary archive logs after the backup is complete.
-l --level	Configures the backup level to 0, 1 or 2. The default value is a full backup (0).
-o --output-file	Specifies the name of the file where progress information is to be outputted.
-S --SA-mode	Performs a backup in standalone mode. The default value is the one specified by the system parameter **CUBRID_MODE**.
-C --CS-mode	Performs a backup in client/server mode. The default value is the one specified by the system parameter **CUBRID_MODE**.
-t	Specifies the maximum number of threads allowed for a parallel

--thread-count	backup. The default value is the number of CPUs in the system.
-z --compress	Performs a compressed backup.
-e --except-active-log	Configures that active log volumes are not included in the backup.
--no-check	Does not perform a consistency check on a database before making a backup.

Performing a backup by specifying the directory in which backup files are to be stored (-D or --destination-path)

The following is an example that uses the **-D** option to store backup files in the specified directory. The backup file directory must be specified before performing this job. If the **-D** option is not specified, backup files are stored in the directory specified in the **databases.txt** file which stores database location information.

```
cubrid backupdb -D /home/cubrid/backup demodb
```

The following example stores backup files in the current directory by using the **-D** option. If you enter a period (.) following the **-D** option as an argument, the current directory is specified.

```
cubrid backupdb -D . demodb
```

Removing archive logs after a backup (-r or --remove-archive)

If the system parameter **media_failure_support** is configured to 1, when the active logs are full, they are written to a new archive log file. If a backup is performed in such a situation and backup volumes are created, backup logs created before the backup will not be used in subsequent backups. The **-r** option is used to remove archive log files that will not be used any more in subsequent backups after the current one is complete.

```
cubrid backupdb -r demodb
```

The **-r** option does not affect the restore because it removes only unnecessary archive logs before the backup, but full restore may not be possible if the administrator removes archive logs created after the backup as well; when you remove archive logs, you must check if those logs would be required in any subsequent restore.

If you perform an incremental backup (backup level 1 or 2) with the **-r** option, there is the risk that normal recovery of the database will be impossible later on. Therefore, it is recommended that the **-r** option only be used when a full backup is performed.

Performing a backup with the backup level specified (-l or --level)

The following example performs an incremental backup of the level specified by using the **-l** option. If the **-l** option is not specified, a full backup is performed. For more information on backup levels, see Incremental Backup.

```
cubrid backupdb -l 1 demodb
```

Saving backup progress information in the specified file (-o or --output-file)

The following example writes the progress of the database backup to the info_backup file by using the **-o** option.

```
cubrid backupdb -o info_backup demodb
```

The following is an example of showing the contents of the info_backup file. You can check the information on the number of threads, compression method, backup start time, the number of permanent volumes, backup progress and backup end time.

```
[ Database(demodb) Full Backup start ]
- num-threads: 1
- compression method: NONE
- backup start time: Mon Jul 21 16:51:51 2008
- number of permanent volumes: 1
- backup progress status
-----------------------------------------------------------------------------
 volume name                  | # of pages | backup progress status    | done
-----------------------------------------------------------------------------
 demodb_vinf                  |          1 | ######################### | done
```

```
 demodb                         |      25000 | ######################### | done
 demodb_lginf                   |          1 | ######################### | done
 demodb_lgat                    |      25000 | ######################### | done
-----------------------------------------------------------------------------
# backup end time: Mon Jul 21 16:51:53 2008
[Database(demodb) Full Backup end]
```

Performing a backup in standalone mode (-S or --SA-mode)

The following example performs a backup in standalone by using the **-S** option. The demodb database is backed up offline. If the **-S** option is not specified, the backup is performed in the mode specified by the **CUBRID_MODE** environment variable.

```
cubrid backupdb -S demodb
```

Performing a backup in client/server mode (-C or --CS-mode)

The following example performs a backup in client/server mode by using the **-C** option. The demodb database is backed up online. If the **-C** option is not specified, a backup is performed in the mode specified by the **CUBRID_MODE** environment variable.

```
cubrid backupdb -C demodb
```

Parallel backup (-t or --thread-count)

The following example performs a parallel backup with the number of threads specified by the administrator by using the **-t** option. Even when the argument of the **-t** option is not specified, a parallel backup is performed by automatically assigning as many threads as CPUs in the system.

```
cubrid backupdb -t 4 demodb
```

Compressed backup (-z or --compress)

The following example compresses the database and stores it in the backup file by using the **-z** option. The size of the backup file and the time required for backup can be reduced by using the **-z** option.

```
cubrid backupdb -z demodb
```

Enabling to exclude active log volumes (-e or --except-active-log)

The following example performs a backup, excluding active logs of the database by using the **-e** option. You can reduce the time required for backup by using the **-e** option. However, extra caution is required because active logs needed for completing a restore to the state of a certain point from the backup point are not included in the backup file, which may lead to an unsuccessful restore.

```
cubrid backupdb -e demodb
```

Disabling a database consistency check (--no-check)

The following example performs a backup without checking the consistency of the database by using the **--no-check** option.

```
cubrid backupdb --no-check demodb
```

Backup Strategy and Method

The following must be considered before performing a backup:

- **Selecting the data to be backed up**
- Determine whether it is valid data worth being preserved.
- Determine whether to back up the entire database or only part of it.
- Check whether there are other files to be backed up along with the database.
- **Choosing a backup method**
- Choose the backup method from one of incremental and online backups. Also, specify whether to use compression backup, parallel backup, and mode.

- Prepare backup tools and devices available.
- **Determining backup time**
- Identify the time when the least usage in the database occur.
- Check the size of the archive logs.
- Check the number of clients using the database to be backed up.

Online Backup

An online backup (or a hot backup) is a method of backing up a currently running database. It provides a snapshot of the database image at a certain point in time. Because the backup target is a currently running database, it is likely that uncommitted data will be saved and the backup may affect the operation of other databases.

To perform an online backup, use the **cubrid backupdb -C** command.

Offline Backup

An offline backup (or a cold backup) is a method of backing up a stopped database. It provides a snapshot of the database image at a certain point in time.

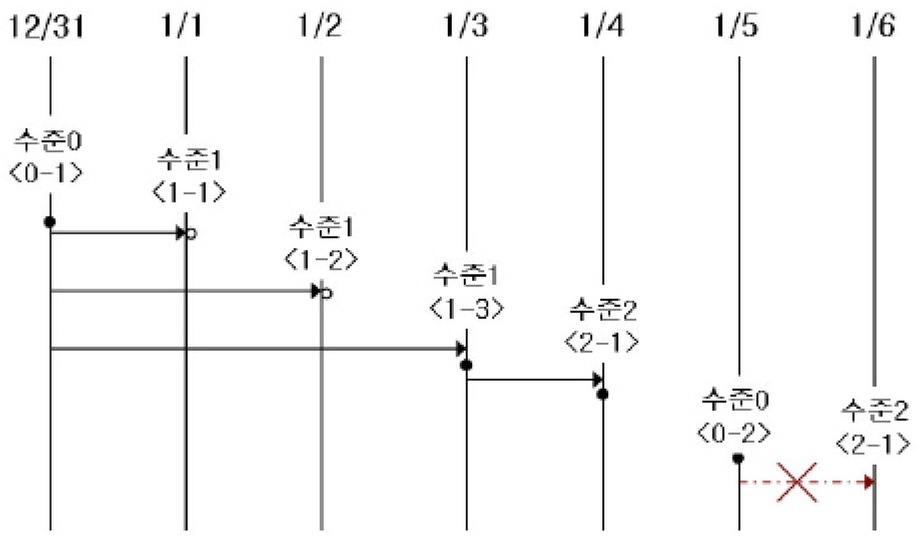
To perform an offline backup, use the **cubrid backupdb -S** command.

Incremental Backup

An incremental backup, which is dependent upon a full backup, is a method of only backing up data that have changed since the last backup. This type of backup has an advantage of requiring less volume and time than a full backup. CUBRID supports backup levels 0, 1 and 2. A higher level backup can be performed sequentially only after a lower lever backup is complete.

To perform an incremental backup, use the **cubrid backupdb -l** <*level*> command.

The following is an example of an incremental backup. With this example, we will examine backup levels in detail.



- **Full backup (backup level 0)** : Backup level 0 is a full backup that includes all database pages.

  The level of a backup which is attempted first on the database naturally becomes a 0 level. **DBA** must perform full backups regularly to prepare for restore situations. In the example, full backups were performed on December 31st and January 5th.

- **First incremental backup (backup level 1)** : Backup level 1 is an incremental backup that only saves changes since the level 0 full backup, and is called a "first incremental backup."

  Note that the first incremental backups are attempted sequentially such as <1-1>, <1-2> and <1-3> in the example, but they are always performed based on the level 0 full backup.

Suppose that backup files are created in the same directory. If the first incremental backup <1-1> is performed on January 1st and then the first incremental backup <1-2> is attempted again on January 2nd, the incremental backup file created in <1-1> is overwritten. The final incremental backup file is created on January 3rd because the first incremental backup is performed again on that day.

Since there can be a possibility that the database needs to be restored the state of January 1st or January 2nd, it is recommended for **DBA** to save the incremental backup files <1-1> and <1-2> separately in storage media before overwriting with the final incremental file.

- **Second incremental backup (backup level 2)** : Backup level 2 is an incremental backup that only saves data that have changed since the first incremental backup, and is called a "second incremental backup."

  A second incremental backup can be performed only after the first incremental backup. Therefore, the second incremental backup attempted on January fourth succeeds; the one attempted on January sixth fails.

  Backup files created for backup levels 0, 1 and 2 may all be required for database restore. To restore the database to its state on January fourth, for example, you need the second incremental backup generated at <2-1>, the first incremental backup file generated at <1-3>, and the full backup file generated at <0-1>. That is, for a full restore, backup files from the most recent incremental backup file to the earliest created full backup file are required.

Compress Backup

A compress backup is a method of backing up the database by compressing it. This type of backup reduces disk I/O costs and saves disk space because it requires less backup volume.

To perform a compress backup, use the **cubrid backupdb -z|--compress** command.

Parallel Backup Mode

A parallel or multi-thread backup is a method of performing as many backups as the number of threads specified. In this way, it reduces backup time significantly. Basically, threads are given as many as the number of CPUs in the system.

To perform a parallel backup, use the **cubrid backupdb -t|--thread-count** command.

Managing Backup Files

One or more backup files can be created in sequence based on the size of the database to be backed up. A unit number is given sequentially (000, 001-0xx) to the extension of each backup file based in the order of creation.

Managing Disk Capacity during the Backup

During the backup process, if there is not enough space on the disk to save the backup files, a message saying that the backup cannot continue appears on the screen. This message contains the name and path of the database to be backed up, the backup file name, the unit number of backup files and the backup level. To continue the backup process, the administrator can choose one of the following options:

- Option 0 : An administrator enters 0 to discontinue the backup.
- Option 1 : An administrator inserts a new disk into the current device and enters 1 to continue the backup.
- Option 2 : An administrator changes the device or the path to the directory where backup files are saved and enters 2 to continue the backup.

```
******************************************************************
Backup destination is full, a new destination is required to continue:
Database Name: /local1/testing/demodb
     Volume Name: /dev/rst1
        Unit Num: 1
    Backup Level: 0 (FULL LEVEL)
Enter one of the following options:
Type
   -  0 to quit.
   -  1 to continue after the volume is mounted/loaded. (retry)
   -  2 to continue after changing the volume's directory or device.
******************************************************************
```

Database Restore

A database restore is the procedure of restoring the database to its state at a certain point in time by using the backup files, active logs and archive logs which have been created in an environment of the same CUBRID version. To perform a database restore, use the **cubrid restoredb** utility or the CUBRID Manager.

The **cubrid restoredb** utility (restordb.exe in Windows) recovers the database from the database backup by using the information written to all the active and archive logs since the execution of the last backup.

Syntax

```
cubrid restoredb [ options ] database_name
[ options ]
-d | -B | -l | -p | -o | -u |
--up-to-date | --backup-file-path | --level | --partial-recovery | --output-file | --use-
database-location-path | --list
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **restoredb** : A command for recovery of the specified database. For a successful recovery, you must prepare backup files, active log files and archive log files. This command can be performed only in standalone mode.
- *options* : A short name option starts with a single dash (-) while a full name option starts with a double dash (--). This option is case sensitive.
- *database_name* : Specifies the name of the database to be recovered.

Return Value

- 0 : Success
- Non-zero : Failure

Option

The following table shows options that can be used with **cubrid restoredb**. Options are case sensitive.

Option	Description
-d **--up-to-date**	Directly sets the time to backup the database or specifies the **backuptime** keyword.
-B **--backup-file-path**	Specifies the directory pathname or device name where backup files are to be located.
-l **--level**	Sets the recovery level to 0, 1 or 2. The default value is a full recovery (0).
-p **--partial-recovery**	Performs a partial recovery.
-o **--output-file**	Specifies the name of the file where recovery information is to be displayed.
-u **--use-database-location-path**	Recovers the database to the path specified in the database location file (**databases.txt**).
--list	Displays information on backup volumes of the database on the screen.

Performing a recovery by specifying a recovery point (-d or --up-to-date)

The following command recovers demodb. If no option is specified, demodb is recovered to the point of the last commit by default. If no active/archive log files are required to recover to the point of the last commit, the database is recovered only to the point of the last backup.

```
cubrid restoredb demodb
```

demodb can be recovered to the given point by using the **-d** option and the syntax which specifies the date and time of the recovery. The user can specify the recovery point manually in the dd-mm-yyyy:hh:mm:ss (e.g. 14-10-2008:14:10:00) format. If no active log/archive log files are required to recover to the point specified, the database is recovered only to the point of the last backup.

```
cubrid restoredb -d 14-10-2008:14:10:00 demodb
```

The following syntax specifies the recovery point by using the **-d** option and the **backuptime** keyword and recovers demodb to the point of the last backup.

```
cubrid restoredb -d backuptime demodb
```

Performing a recovery by specifying the directory path to the backup files (-B or --backup-file-path)

You can specify the directory where backup files are to be located by using the **-B** option. If this option is not specified, the system retrieves the backup information file (**dbname_bkvinf**) generated upon a database backup; the backup information file in located in the **log_path** directory specified in the database location information file (**databases.txt**). And then it searches the backup files in the directory path specified in the backup information file. However, if the backup information file has been damaged or the location information of the backup files has been deleted, the system will not be able to find the backup files. Therefore, the administrator must manually specify the directory where the backup files are located by using the **-B** option.

```
cubrid restoredb -B /home/cubrid/backup demodb
```

If the backup files of demodb is in the current directory, the administrator can specify the directory where the backup files are located by using the **-B** option.

```
cubrid restoredb -B . demodb
```

Performing a recovery by specifying the backup level (-l or --level)

You can perform a restoration by specifying the backup level of the database to 0, 1, or 2. For more information on backup levels, see Increment Backup.

```
cubrid restoredb -l 1 demodb
```

Performing a partial recovery (-p or --partial-recovery)

The following command performs a partial recovery without requesting for the user's response by using the **-p** option. If active or archive logs written after the backup point are not complete, by default the system displays a request message informing that log files are needed and prompting the user to enter an execution option. A partial recovery can be performed directly without such a request message by using the **-p** option. Therefore, if the **-p** option is used when performing a recovery, data is always recovered to the point of the last backup.

```
cubrid restoredb -p demodb
```

When the **-p** option is not specified, the message requesting the user to select the execution option is as follows:

```
***********************************************************
Log Archive /home/cubrid/test/log/demodb_lgar002
 is needed to continue normal execution.
   Type
   -  0 to quit.
   -  1 to continue without present archive. (Partial recovery)
   -  2 to continue after the archive is mounted/loaded.
   -  3 to continue after changing location/name of archive.
***********************************************************
```

- Option 0 : An administrator enters 0 to stop the recovery.
- Option 1 : An administrator enters 1 to perform a partial recovery without log files.
- Option 2 : An administrator enters 2 to perform a recovery after moving archive logs to the current device.
- Option 3 : An administrator enters 3 after changing a log location to resume a restoration.

Storing recovery progress information in the specified file (-o or --output-file)

The following command writes the recovery progress of the database to the info_restore file by using the **-o** option.

```
cubrid restoredb -o info_restore demodb
```

Recovering data to the directory specified in the database location file (-u or --use-database-location-path)

The following syntax recovers the database to the path specified in the database location file (**databases.txt**) by using the **-u** option. The **-u** option is useful when you perform a backup on server A and recover the backup files on server B.

```
cubrid restoredb -u demodb
```

Checking the backup information of the database (--list)

The following syntax displays the information on backup files of the database by using the **--list** option; it does not perform recovery.

```
cubrid restoredb --list demodb
```

The following is an example of backup information displayed as a result of using the **--list** option. You can identify the path to which backup files of the database are originally stored as well as backup levels.

```
*** BACKUP HEADER INFORMATION ***
Database Name: /local1/testing/demodb
 DB Creation Time: Mon Oct 1 17:27:40 2008
         Pagesize: 4096
Backup Level: 1 (INCREMENTAL LEVEL 1)
        Start_lsa: 513|3688
         Last_lsa: 513|3688
Backup Time: Mon Oct 1 17:32:50 2008
 Backup Unit Num: 0
Release: 8.1.0
     Disk Version: 8
Backup Pagesize: 4096
Zip Method: 0 (NONE)
        Zip Level: 0 (NONE)
Previous Backup level: 0 Time: Mon Oct 1 17:31:40 2008
(start_lsa was -1|-1)
Database Volume name: /local1/testing/demodb_vinf
     Volume Identifier: -5, Size: 308 bytes (1 pages)
Database Volume name: /local1/testing/demodb
     Volume Identifier: 0, Size: 2048000 bytes (500 pages)
Database Volume name: /local1/testing/demodb_lginf
     Volume Identifier: -4, Size: 165 bytes (1 pages)
Database Volume name: /local1/testing/demodb_bkvinf
     Volume Identifier: -3, Size: 132 bytes (1 pages)
```

With the backup information displayed by using the **--list** option, you can check that backup files have been created at the backup level 1 as well as the point where the full backup of backup level 0 has been performed. Therefore, to recover the database in the example, you must prepare backup files for backup levels 0 and 1.

Restore Strategy and Procedure

The following must be considered before performing a database restore:

- **Preparing a backup file**
- Identify the directory where the backup and log files are to be stored.
- If the database has been incrementally backed up, check whether an appropriate backup file for each backup level exists.
- Check whether the backed-up CUBRID database and the CUBRID database to be backed up are the same version.
- **Choosing a restore method**
- Determine whether to perform a partial or full restore.
- Determine whether or not to perform a restore using incremental backup files.
- Prepare restore tools and devices available.
- **Determining restore time**
- Identify the point in time when the database server was terminated.
- Identify the point in time when the last backup was performed before database failure.
- Identify the point in time when the last commit was made before database failure.

Database Restore Procedure

The following is an example of a backup and restore process described in the order of time.

- Performs a full backup of demodb which stopped running at 2008/8/14 04:30.
- Performs the first incremental backup of demodb running at 2008/8/14 10:00.
- Performs the first incremental backup of demodb running at 2008/8/14 15:00. Overwrites the first incremental backup file in step 2.
- A system failure occurs at 2008/8/14 15:30, and the system administrator prepares the restore of demodb. Sets the restore time as 15:25, which is the time when the last commit was made before database failure
- The system administrator prepares the full backup file created in Step 1 and the first incremental backup file created in Step 3, restores the demodb database up to the point of 15:00, and then prepares the active and archive logs to restore the database up to the point of 15:25.

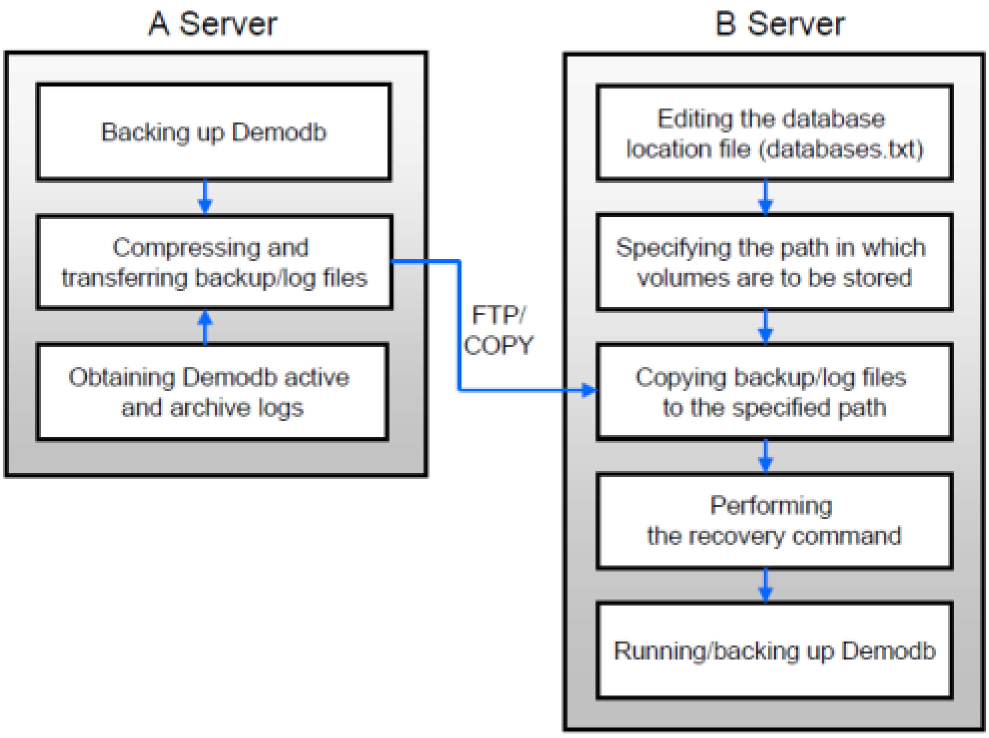
Time	Command	Description
2008/8/14 04:25	cubrid server stop demodb	Shuts down demodb.
2008/8/14 04:30	cubrid backupdb -S -D /home/backup -l 0 demodb	Performs a full backup of demodb in offline mode and creates backup files in the specified directory.
2008/8/14 05:00	cubrid server start demodb	Starts demodb.
2008/8/14 10:00	cubrid backupdb -C -D /home/backup -l 1 demodb	Performs the first incremental backup of demodb online and creates backup files in the specified directory.
2008/8/14 15:00	cubrid backupdb -C -D /home/backup -l 1 demodb	Performs the first incremental backup of demodb online and creates backup files in the specified directory. Overwrites the first incremental backup file created at 10:00.
2008/8/14 15:30		A system failure occurs.
2008/8/14 15:40	cubrid restoredb -l 1 -d 08/14/2008:15:25:00 demodb	Restores demodb based on the full backup file, first incremental backup file, active logs and archive logs. The database is restored to the point of 15:25 by the full and first incremental backup files, the active and archive logs.

Restoring Database to Different Server

The following shows how to back up demodb on server A and restore it on server B with the backed up files.

Backup and Restore Environments

Suppose that demodb is backed up in the /home/cubrid/db/demodb directory on server A and restored into /home/cubrid/data/demodb on server B.



1. Backing up on server A

   Back up demodb on server A. If a backup has been performed earlier, you can perform an incremental backup for data only that have changed since the last backup. The directory where the backup files are created, if not specified in the **-D** option, is created by default in the location where the log volume is stored. The following is a backup command with recommended options. For more information on the options, see Database Backup.

   ```
   cubrid backupdb -z -t demodb
   ```

2. Editing the database location file on Server B

   Unlike a general scenario where a backup and restore are performed on the same server, in a scenario where backup files are restored using a different server, you need to add the location information on database restore in the database location file (**databases.txt**) on server B. In the diagram above, it is supposed that **demodb** is restored in the **/home/cubrid/data/demodb** directory on server B (hostname: pmlinux); edit the location information file accordingly and create the directory on server B.

   Put the database location information in one single line. Separate each item with a space. The line should be written in [database name]. [data volume path] [host name] [log volume path] format; that is, write the location information of **demodb** as follows:

   ```
   demodb /home/cubrid/data/demodb pmlinux /home/cubrid/data/demodb
   ```

3. Transferring backup/log files to server B

   For a restore, you must prepare a backup file (e.g. demodb_bk0v000) and a backup information file (e.g. demodb_bkvinf) of the database to be backed up. To restore the entire data up to the point of the last commit, you must prepare an active log (e.g. demodb_lgat) and an archive log (e.g. demodb_lgar000). Then, transfer the backup information, active log, and archive log files created on server A to server B. That is, the backup information, active log and archive log files must be located in a directory (e.g. /home/cubrid/temp) on server B.

4. Restoring the database on server B

   Perform database restore by calling the **cubrid restoredb** utility from the directory into which the backup, backup information, active log and archive log files which were transferred to server B had been stored. With the **-u** option, demodb is restored in the directory path from the **databases.txt** file.

   ```
   cubrid restoredb -u demodb
   ```

   To call the **cubrid restoredb** utility from a different path, specify the directory path to the backup file by using the **-B** option as follows:

   ```
   cubrid restoredb -u -B /home/cubrid/temp demodb
   ```

5. Backing up the restored database on server B

   Once the restore of the target database is complete, run the database to check if it has been properly restored. For stable management of the restored database, it is recommended to restore the database again on the server B environment.

CUBRID HA

Overview

CUBRID HA

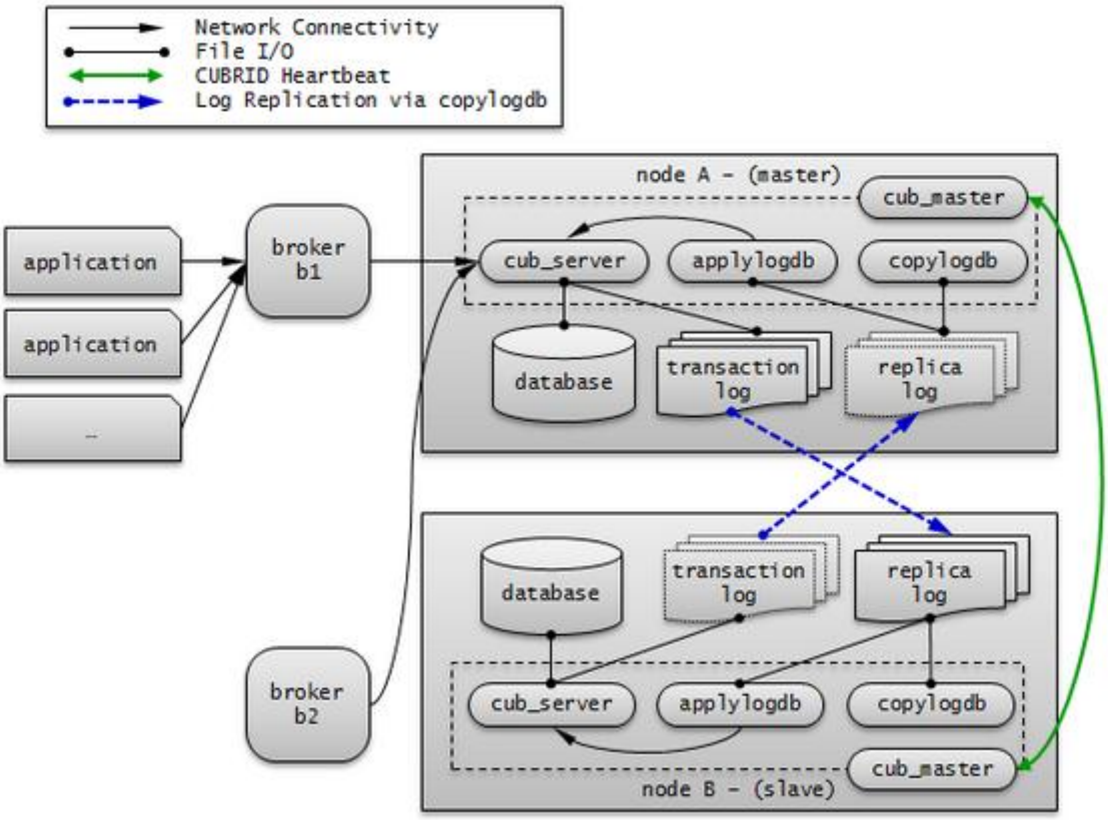
High Availability (HA) refers to a feature to provide uninterrupted service in the event of hardware, software, or network failure. This ability is a critical element in the network computing area where services should be provided 24/7. An HA system consists of more than two server systems, each of which provides uninterrupted services, even when a failure occurs in one of them.

CUBRID HA is an implementation of High Availability. The CUBRID HA feature ensures database synchronization among multiple servers when providing service. When an unexpected failure occurs in the system which is operating services, this feature minimizes the service down time by allowing the other system to carry out the service automatically.

The CUBRID HA feature is in a shared-nothing structure. To synchronize data from an active server to a standby server, the CUBRID HA feature executes the following two steps.

- Transaction log multiplexing: Replicates the transaction logs created by an active server to another node in real time.
- Transaction log reflection: Analyzes replicated transaction logs in real time and reflects the data to a standby server.

The CUBRID HA feature executes the steps described above in order to always maintain data synchronization between an active server and a standby server. For this reason, if an active server is not working properly because of a failure occurring in the master node that had been providing service, the standby server of the slave node provides service instead of the failed server. The CUBRID HA feature monitors the status of the system and CUBRID in real time. It uses heartbeat messages to execute an automatic failover when a failure occurs.



CUBRID HA Concept

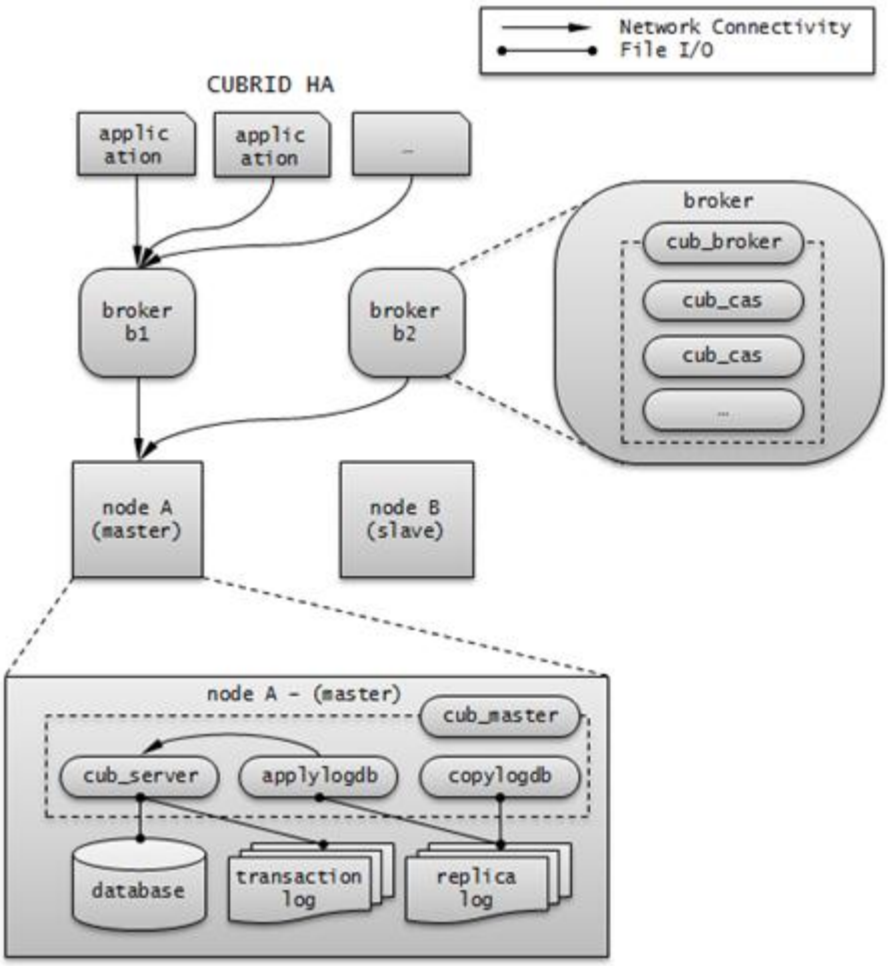
Groups and Nodes

A node is a logical unit that makes up CUBRID HA. It can become one of the following nodes according to its status: master node, slave node, or replica node.

- **Master node** : A node to be replicated. It provides all services which are read, write, etc. using an active server.
- **Slave node** : A node that has the same information as a master node. Changes made in the master node are automatically reflected to the slave node. It provides the read service using a standby server, and a failover will occur when the master node fails.
- **Replica node** : A node that has the same information as a master node. Changes made in the master node are automatically reflected to the replica node. It provides the read service using a standby server, and no failover will occur when the master node fails.

The CUBRID HA group consists of the nodes described above. You can configure the members of this group by using the **ha_node_list** and **ha_replica_list** in the **cubrid.conf** file. Nodes in a group have the same information. They exchange status checking messages periodically and a failover will occurs when the master node fails.

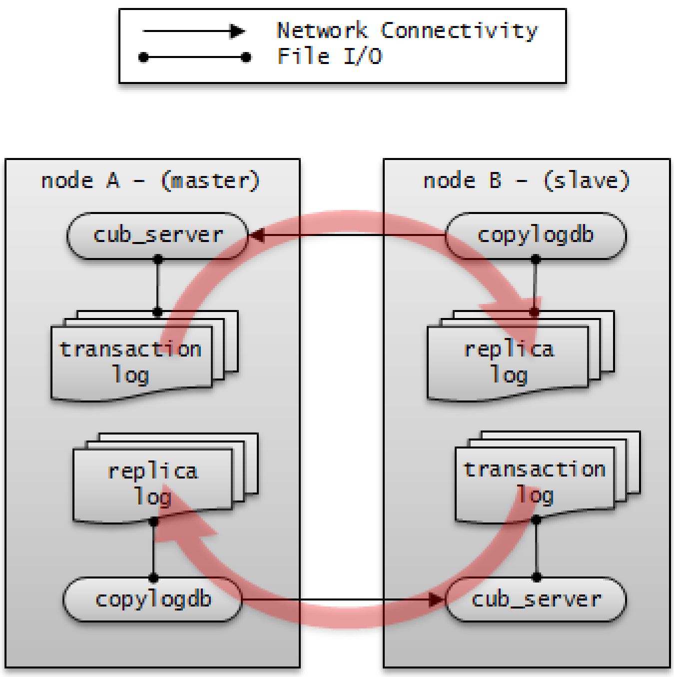
A node includes the master process (cub_master), the database server process (cub_server), the replication log copy process (copylogdb), the replication log reflection process (applylogdb), etc.



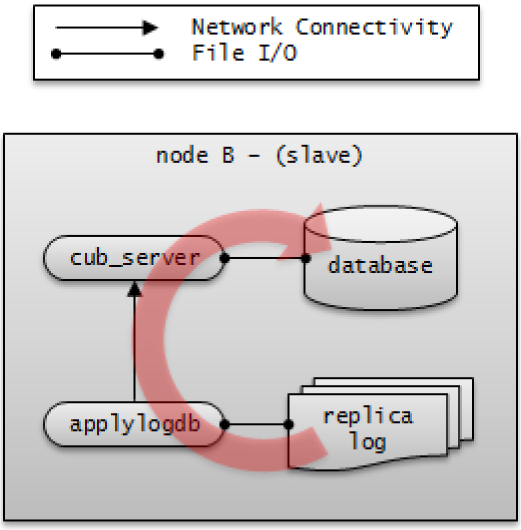
Processes

A CUBRID HA node consists of one master process (cub_master), one or more database server processes (cub_server), one or more replication log copy processes (copylogdb), and one or more replication log reflection processes (applylogdb). When a database is configured, database server processes, replication log copy processes, and replication log reflection processes will start. Because copy and reflection of a replication log are executed by different processes, the delay in replicating reflections does not affect the transaction that is being executed.

- **Master process (cub_master)** : Exchanges heartbeat messages to control the internal management processes of CUBRID HA.
- **Database server process (cub_server)** : Provides services such as read or write to the user. For more information, see Server.
- **Replication log copy process (copylogdb)** : Copies all transaction logs in a group. When the replication log copy process requests a transaction log from the database server process of the target node, the database server process returns the corresponding log. The location of copied transaction logs can be configured in the **REPL_LOG_HOME** of **cubrid-ha**. Use cubrid applyinfo utility to verify the information of copied replication logs. The replication log copy process has following three modes: SYNC, SEMISYNC, and ASYNC. You can configure it with the **LW_SYNC_MODE** of **cubrid-ha**. For more information on these modes, see Multiplexing Logs.



- **Replication log reflection process (applylogdb)** : Reflects the log that has been copied by the replication log copy process to a node. The information of reflected replications is stored in the internal catalog (db_ha_apply_info). You can use the cubrid applyinfo utility to verify this information.

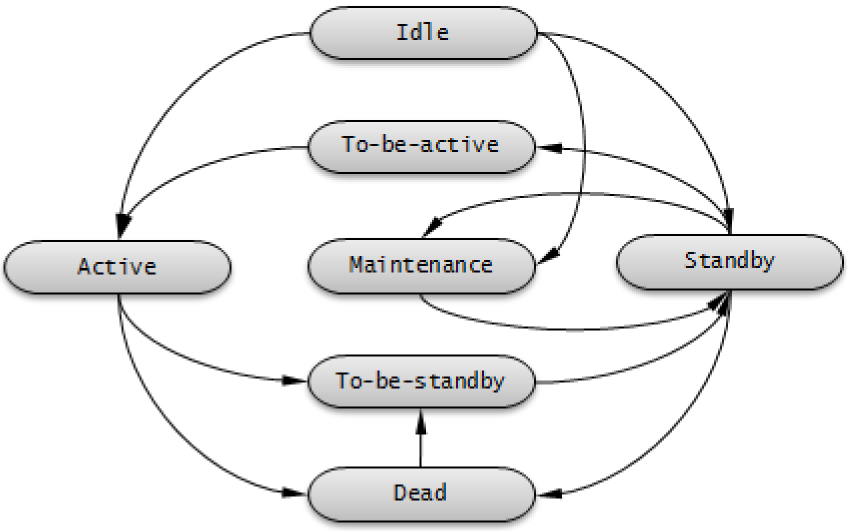


Servers

Here, the word "server" is a logical representation of database server processes. Depending on its status, a server can be either an active server or a standby server.

- **Active server** : A server that belongs to a master node; the status is active. An active server provides all services, including read, write, etc. to the user.
- **Standby server**: A standby server that belongs to a non-master node; the status is standby. A standby server provides only the read service to the user.

The server status changes based on the status of the node. You can use the cubrid changemode utility to verify server status. The maintenance mode exists for operational convenience and you can change it by using the **cubrid changemode** utility.



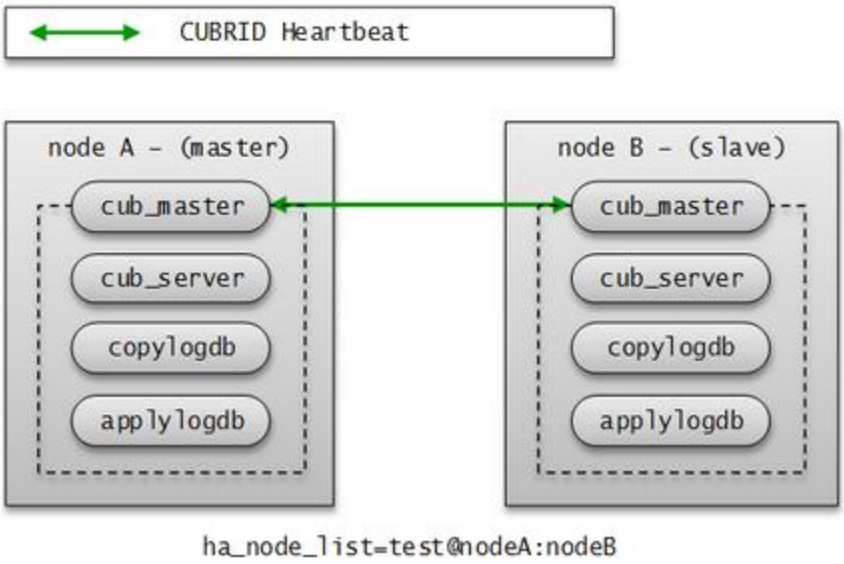
- **active** : The status of servers that run on a master node is usually active. In this status, all services including read, write, etc. are provided.
- **standby** : The status of servers that run on a slave node or a replica node is standby. In this status, only the read service is provided.
- **maintenanc** : The status of servers can be manually changed for operational convenience is maintenance. In this status, only a csql can access and no service is provided to the user.

- **to-be-active** : The status in which a standby server will become active for reasons such as failover, etc. is to-be-active. In this status, servers prepare to become active by reflecting transaction logs from the existing master node to its own server.
- Other : This status internally used.

heartbeat Message

As a core element to provide the HA feature, it is a message exchanged among master, slave, and replica nodes to monitor the status of other nodes. A master process periodically exchanges heartbeat messages with all other master processes in the group. A heartbeat message is exchanged through the UDP port configured in the **ha_port_id** parameter of **cubrid.conf**. The exchange interval of heartbeat messages is determined by an internally configured value.

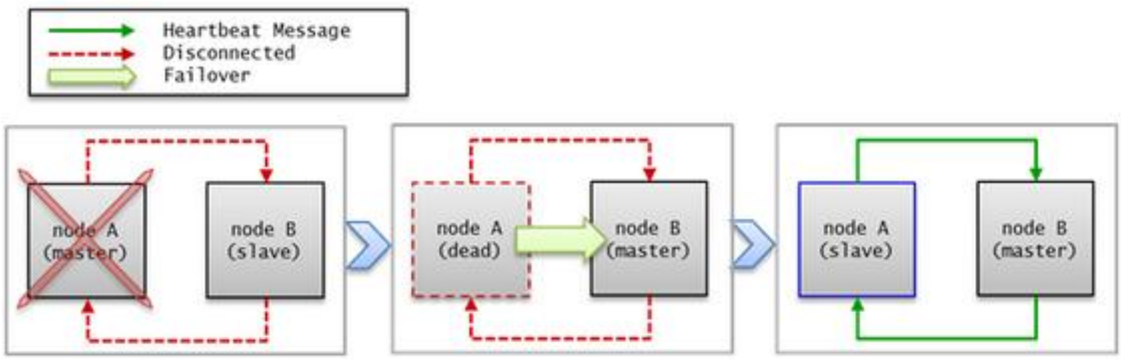
When the master node fails, a failover occurs to a slave node.



Failover and Failback

A failover means that the highest priority slave node automatically becomes a new master node when the original master node fails to provide services due to a failure. A master process calculates scores for all nodes in the CUBRID HA group based on the collected information, promotes slave nodes to master modes when it is necessary, and then notifies the management process of the changes it has made.

A failback means that the previously failed master node automatically becomes a master node back after the failure node is restored. The CUBRID HA does not currently support this functionality.



If a heartbeat message fails to deliver, a failover will occur. For this reason, servers with unstable connection may experience frequent failovers even though no actual failures occur. To prevent a failover from occurring in the situation described above, configure **ha_ping_ports**. Configuring **ha_ping_ports** will send a ping message to a node specified in

ha_ping_ports in order to verify whether the network is stable or not when a heartbeat message fails to deliver. For more information on configuring **ha_ping_ports**, see cubrid_ha.conf.

Broker Mode

A broker can access a server with one of the following modes: **Read Write**, **Read Only**, **Slave Only**, or **Preferred Host Read Only**. This configuration value is determined by a user.

A broker finds and connects to a suitable server by trying to establish a connection in the order of server connections; this is, if it fails to establish a connection, it tries another connection to the next server defined until it reaches the last server. If no connection is made even after trying all servers, the broker fails to connect to a server.

For more information on how to configure broker mode, see cubrid_broker.conf.

Read Write

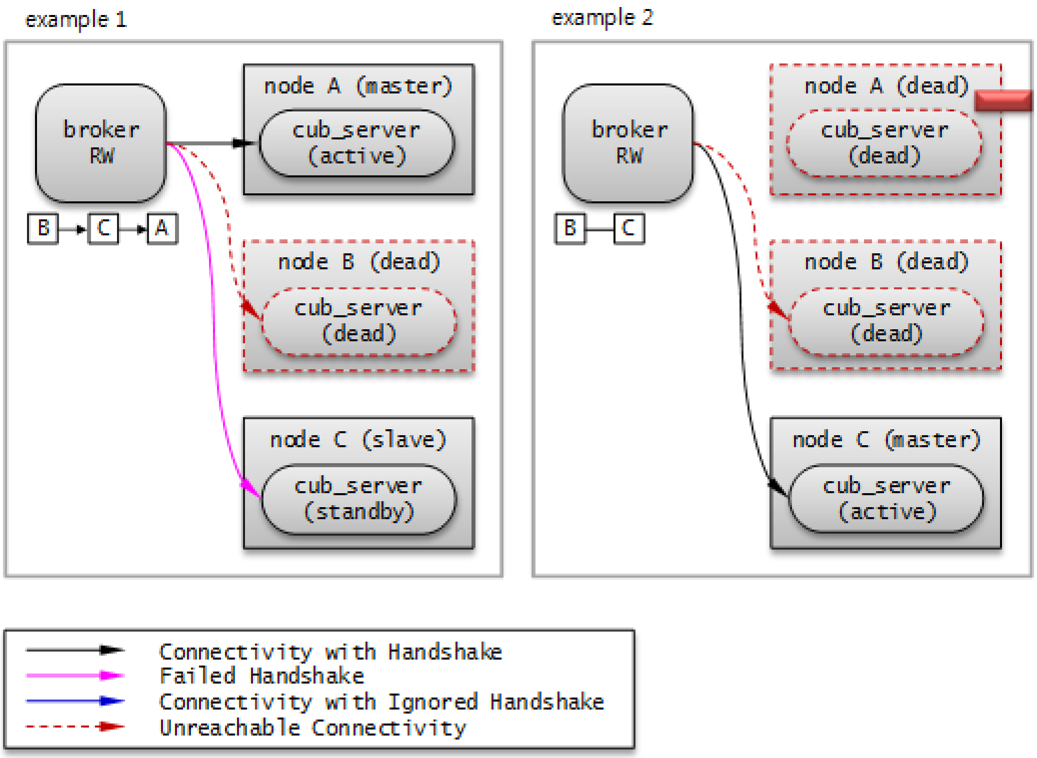
A broker that provides read and write services. This broker is usually connected to an active server. If no active servers exist, this broker will be connected to a standby server. For this reason, a Read Write broker can be temporarily connected to a standby server.

When the broker temporarily establishes a connection to a standby server, it will disconnect itself from the standby server at the end of every transaction so that it can attempt to find an active server at the beginning of the next transaction. When it is connected to the standby server, only read service is available. Any write requests will result in a server error.

The order of server connection is described below:

- The broker tries to establish a connection to an existing server connected (if exsits). The active status of the server means the connection is complete.
- The broker tries to establish a connection to the hosts specified in the databases.txt file in a sequence. The active status of the server means the connection is complete.
- The broker tries to establish a connection to the hosts specified in the databases.txt file in a sequence and connects to the first available host.

databases.txt			
#db-name	vol-path	db-host	log-path
tdb	/home/db	node B:node C:node A	/home/db/log



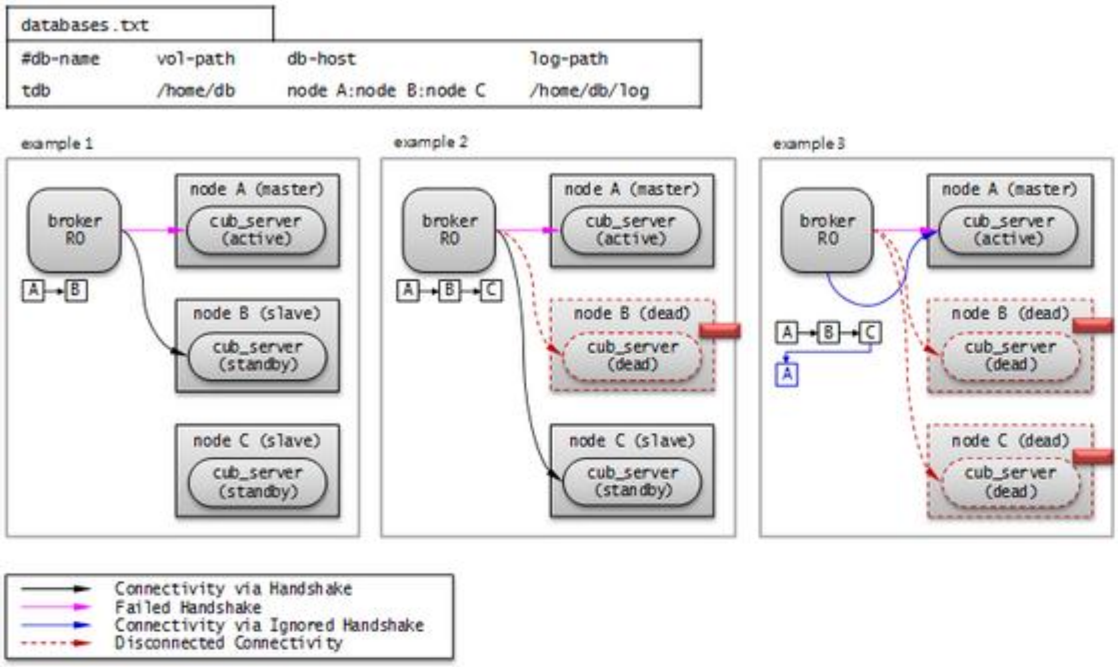
Read Only

A broker that provides the read service. This broker is connected to a standby server if possible. For this reason, a Read Only broker can be connected to an active server temporarily.

Once it establishes a connection with an active server, it will maintain that connection even if a standby server exists. To disconnect from the active server and reconnect to a standby server, you should execute the **cubrid_broker reset** command. An error will occur when a Read Only broker receives a write request; therefore, only the read service will be available even if it is connected to an active server.

The order of server connection is described below:

- The broker tries to establish a connection to an existing server connected (if exsits). The standby status of the server means the connection is complete.
- The broker tries to establish a connection to the hosts specified in the databases.txt file in a sequence. The standby status of the server means the connection is complete.
- The broker tries to establish a connection to the hosts specified in the databases.txt file in a sequence and connects to the first available host.

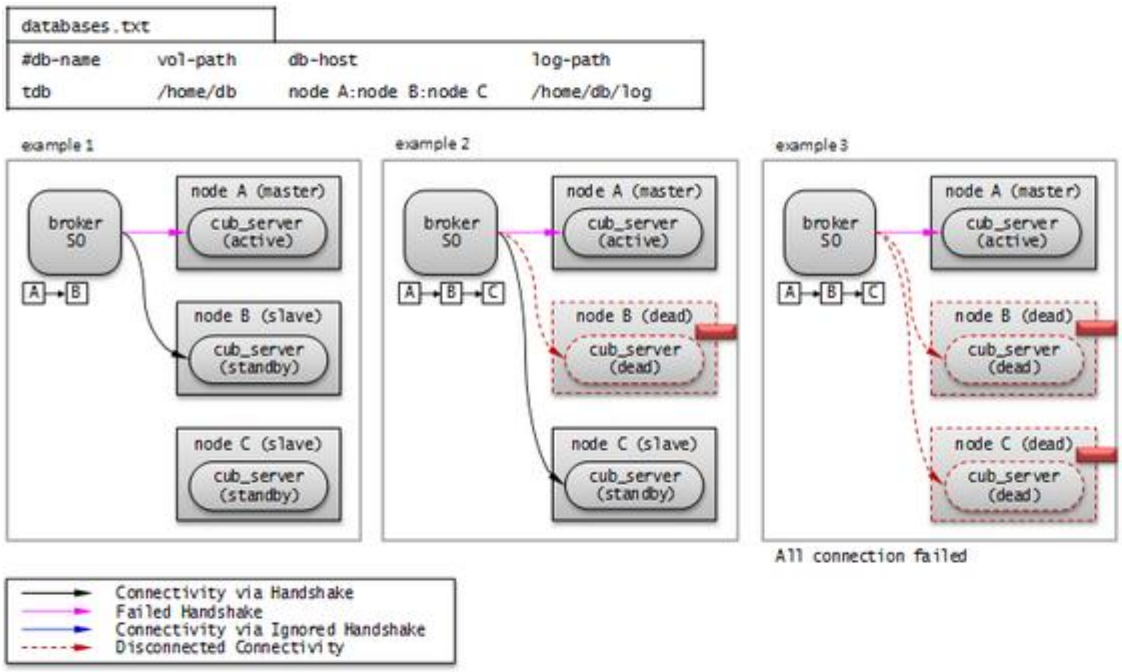


Slave Only

A broker that provides the read service. This broker can only be connected to a standby server. If no standby server exists, no service will be provided.

The order of server connection is described below:

- The broker tries to establish a connection to an existing server connected (if exists). The standby status of the server means the connection is complete.
- The broker tries to establish a connection to the hosts specified in the databases.txt file in a sequence. The standby status of the server means the connection is complete.



Preferred Host Read Only

A broker that provides the read service. This works in the same manner as the Read Only broker except for its server connection order and server selecting criteria. The server connection order and server selecting criteria can be configured in **PREFERRED_HOSTS**. For more information on configuring these, see cubrid_broker.conf.

The order of server connection is described below:

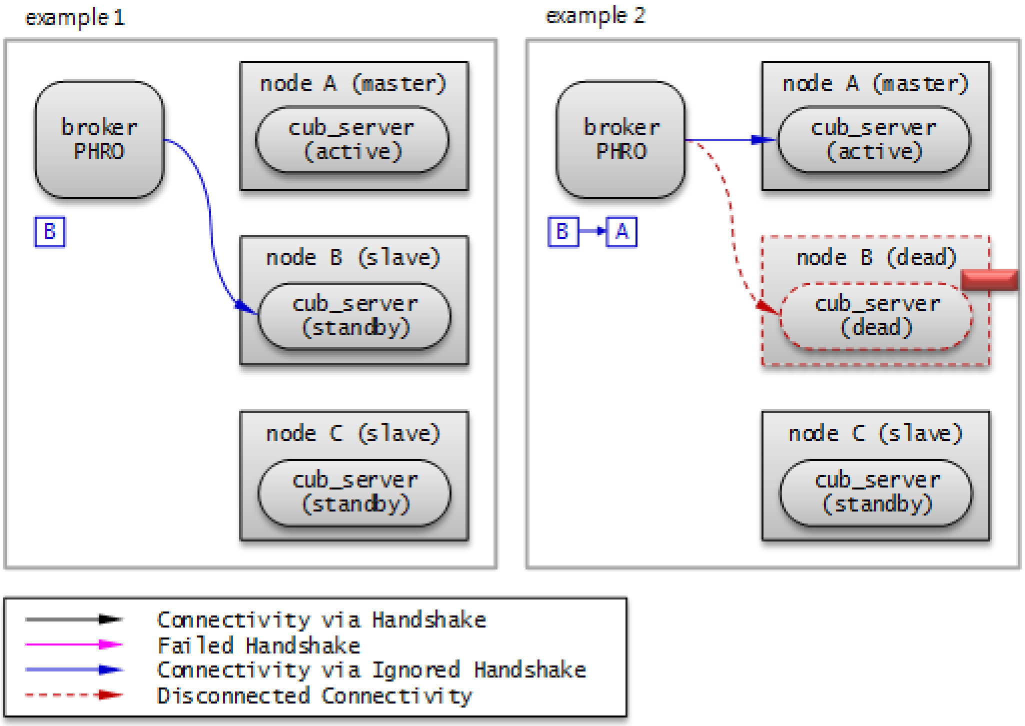
- The broker tries to establish a connection to the hosts specified in PREFERRED_HOSTS in a sequence and connects to the first available host.
- The broker tries to establish a connection to the hosts specified in the databases.txt file in a sequence. The standby status of the server means the connection is complete.
- The broker tries to establish a connection to the hosts specified in the databases.txt file in a sequence and connects to the first available host.

databases.txt

#db-name	vol-path	db-host	log-path
tdb	/home/db	node A:node B:node C	/home/db/log

cubrid_broker.conf

```
PERFERRED_HOSTS=node B:node A
```



CUBRID HA Feature

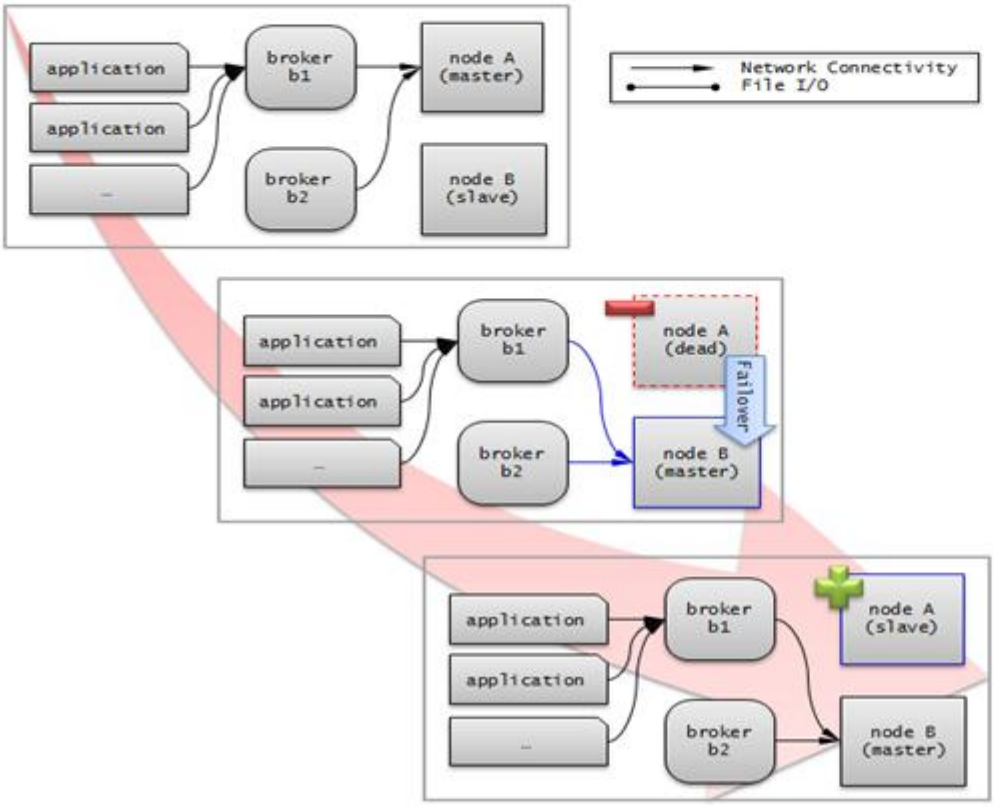
Server Duplexing

Duplexing servers is building a system by configuring duplicate hardware equipment to provide the CUBRID HA feature. This method will prevent any interruptions in a server in case of occurring a hardware failure.

Server failover

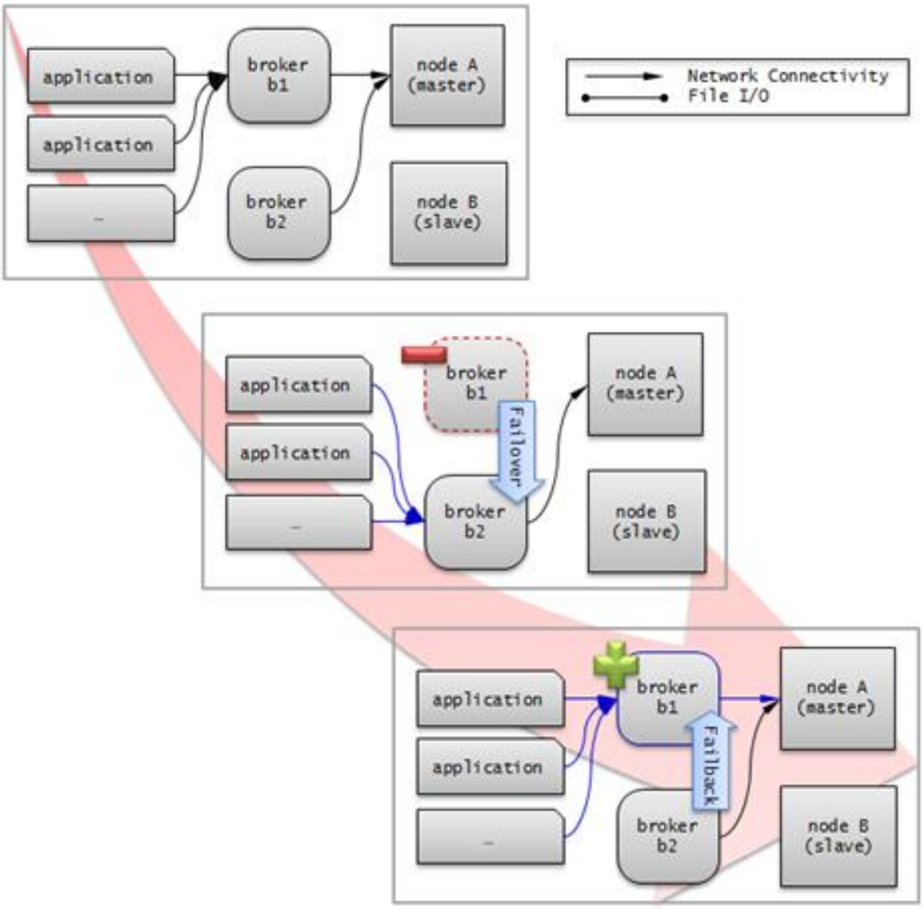
A broker defines server connection order and connects to a server according to the defined order. If the connected server fails, the broker connects to the server with the next highest priority. This requires no processing in the application side.

The actions taken when the broker connects to another server may differ according to the current mode of the broker. For more information on the server connection order and configuring broker mode, see cubrid_broker.conf.



Broker Duplexing

As a 3-tier DBMS, CUBRID has middleware called the broker which relays applications and database servers. To provide the HA feature, the broker also requires duplicate hardware equipment. This method will prevent any interruptions in a broker in case of occurring a hardware failure.



To use this feature, you must define multiple-broker connections in the connection URL of JDBC. For more information, see JDBC Configuration.

Broker failover

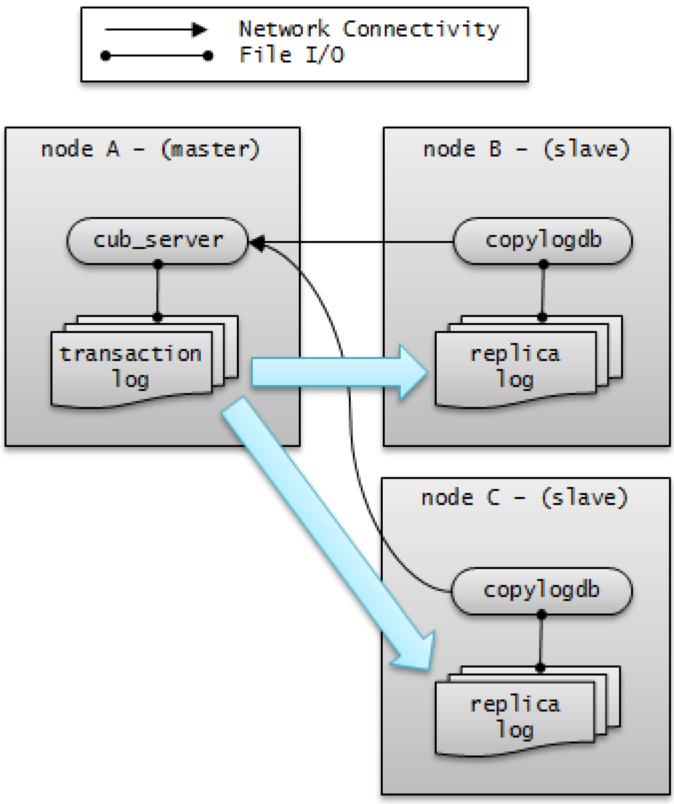
JDBC tries to connect to a broker that has the highest priority in the connection URL. If a failure occurs, it tries to connect to the next broker defined in the order of priority of the connection URL. This requires no processing in the application side as it is processed within the JDBC driver.

Broker failback

If the failed broker is recovered after a failover, the connection to the existing broker is terminated and a new connection is established with the recovered broker which has the highest priority. This requires no processing in the application side as it is processed within the JDBC driver. Exeuction time of failback depends on the value configured in JDBC connection URL. For more information, see JDBC Configuration.

Log Multiplexing

CUBRID HA keeps every node in the CUBRID HA group with the identical structure by copying and reflecting transaction logs to all nodes included in the CUBRID HA group. As the log copy structure of CUBRID HA is a mutual copy between the master and the slave nodes, it has a disadvantage of increasing the size of a log volume. However, it has an advantage of flexibility in terms of configuration and failure handling, comparing to the chain-type copy structure.



The transaction log copy modes include **SYNC**, **SEMISYNC**, and **ASYNC**. This value can be configured by the user in [cubrid_ha.conf]() file.

SYNC Mode

When transactions are committed, the created transaction logs are copied to the slave node and stored as a file. The transaction commit is complete after receiving a notice on its success. Although the time it takes to execute commit in this mode may be longer than that in other modes, this is the safest method because the copied transaction logs are always guaranteed to be reflected to the standby server even if a failover occurs.

SEMISYNC Mode

When transactions are committed, the created transaction logs are copied to the slave node and stored as a file according to the internally optimized interval. The transaction commit is complete after receiving a notice of its success. The committed transactions in this mode are guaranteed to be reflected to the slave node sometime in the future.

Because SEMISYNC mode does not always store replication logs as a file, the execution time of commit can decrease, comparing to the SYNC mode. However, data synchronization between nodes may be delayed because replication logs are not reflected until it is stored as a file.

ASYNC Mode

When transactions are committed, the created transaction logs are sent to the slave node and complete. Because whether or not logs being stored in a slave node are not verified in this mode, it is not guaranteed that the committed transactions are reflected to the slave node.

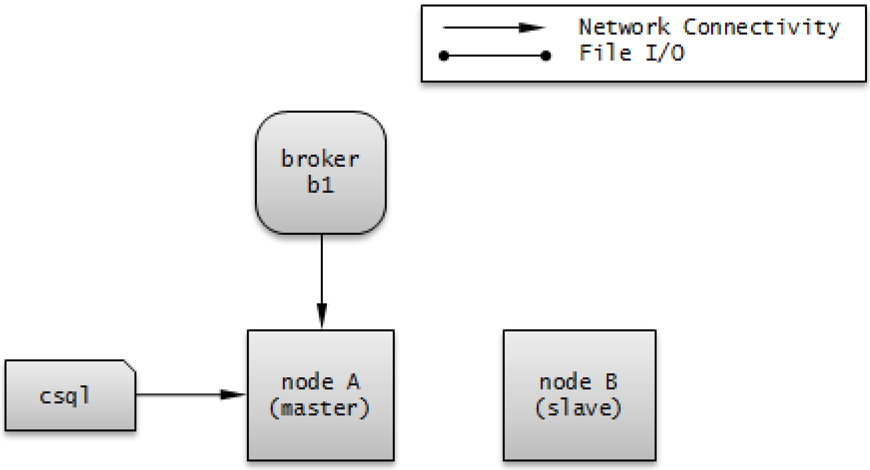
Although ASYNC mode provides a better performance as it has almost no delay when executing commit, there may be data inconsistency in its nodes.

Quick Start

Preparation

Structure Diagram

The diagram below aims to help users who are new to CUBRID HA, by explaining a simple procedure of the CUBRID HA configuration.



Specifications

Linux and CUBRID 2008 version R2.2 or higher must be installed on the equipment to be used as the master and the slave nodes.

Specifications of Configuring the CUBRID HA Equipment

	CUBRID Version	OS
For master nodes	CUBRID 2008 R2.2 or higher	Linux
For slave nodes	CUBRID 2008 R2.2 or higher	Linux

Creating Databases and Configuring Servers

Creating Databases

Create databases to be included in CUBRID HA at each node of the CUBRID HA in the same manner. Modify the options for database creation as needed.

```
[master]$ cd $CUBRID_DATABASES
[master]$ mkdir demodb
[master]$ cd demodb
[master]$ mkdir log
[master]$ cubrid createdb -L ./log demodb
Creating database with 5000 pages.

CUBRID 2008 R4.0

[master]$
```

Configuring cubrid.conf

Ensure **ha_mode** of **$CUBRID/conf/cubrid.conf** in every CUBRID HA node has the same value.

```
# Enable CUBRID HA
ha_mode=on
```

Configuring cubrid_ha.conf

Ensure **ha_port_id**, **ha_node_list**, and **ha_db_list** of **$CUBRID/conf/cubrid_ha.conf** in every

CUBRID HA node have the same value.

```
ha_port_id=12345
ha_node_list=cubrid@nodeA:nodeB
ha_db_list=testdb1:testdb2
```

Starting and Verifying CUBRID HA

Starting CUBRID HA

Execute the **cubrid heartbeat start** at each node in the CUBRID HA group. Note that the node executing **cubrid heartbeat start** first will become a master node.

- Master node

```
[master]$ cubrid heartbeat start
```

- Slave node

```
[slave]$ cubrid heartbeat start
```

Verifying CUBRID HA Status

Execute **cubrid heartbeat status** at each node in the CUBRID HA group to verify its configuration status.

```
[master]$ cubrid heartbeat status
@ cubrid heartbeat list
 HA-Node Info (current master-node-name, state master)
   Node slave-node-name (priority 2, state slave)
   Node master-node-name (priority 1, state master)
 HA-Process Info (master 9289, state master)
   Applylogdb demodb@localhost:/home1/cubrid1/DB/demodb_slave.cub (pid 9423, state
registered)
   Copylogdb demodb@slave-node-name:/home1/cubrid1/DB/demodb_slave.cub (pid 9418, state
registered)
   Server demodb (pid 9306, state registered_and_active)

[master]$
```

Use the **cubrid changemode** utility at each node in the CUBRID HA group to verify the status of the server.

- Master node

```
[master]$ cubrid changemode demodb@localhost
The current HA running mode of the server 'demodb@localhost' is active.
```

- Slave node

```
[slave]$ cubrid changemode demodb@localhost
The current HA running mode of the server 'demodb@localhost' is standby.
```

Verifying the CUBRID HA Operation

Ensure the change has been correctly reflected to the standby server of the slave node after writing the active server of the master node. Note that a primary key must exist when creating a table.

- Master node

```
[master]$ csql -u dba demodb@localhost -c "create table abc(a int, b int, c int,
primary key(a));"
[master]$ csql -u dba demodb@localhost -c "insert into abc values (1,1,1);"
[master]$
```

- Slave node

```
[slave]$ csql -u dba demodb@localhost -l -c "select * from abc;"
=== <Result of SELECT Command in Line 1> ===
<00001> a: 1
        b: 1
        c: 1
[slave]$
```

Configuring and Starting Broker, and Verifying the Broker Status

Configuring the Broker

To provide normal service during a server failover, it is necessary to configure an available node in the **db-host** of **databases.txt**. If the user does not specify it, the **ACCESS_MODE** of the broker will be configured to Read Write mode by default.

```
#db-name        vol-path                db-host         log-path
demodb          /home1/cubrid1/DB/demodb nodeA:nodeB       /home1/cubrid1/DB/demodb/log
```

Starting Broker and Verifying its Status

Because the broker is for the JDBC or CCI application to access, you do not need to start the broker for a simple CUBRID HA test. You can start the broker with **cubrid broker start** and stop it with **cubrid broker stop**.

The following is an example of executing the broker from the master node.

```
[master]$ cubrid broker start
@ cubrid broker start
++ cubrid broker start: success
[master]$ cubrid broker status
@ cubrid broker status
% query_editor OFF
% demodb  - cub_cas [9531,33000] /home1/cubrid1/CUBRID/log/broker//demodb.access
/home1/cubrid1/CUBRID/log/broker//demodb.err
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60.00, LONG_QUERY_TIME:60.00, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
---------------------------------------------------------
ID   PID   QPS   LQS PSIZE STATUS
---------------------------------------------------------
 1  9532     0     0  48120  IDLE
```

Environment Configuration

cubrid.conf

The **cubrid.conf** file that has general information on configuring CUBRID is located in the **$CUBRID/conf** directory. This section explains the **cubrid.conf** parameters used by CUBRID HA.

ha_mode

A parameter used to configure whether to use CUBRID HA feature. The default value is **off**.

- **off** : CUBIRD HA feature is not used.
- **on** : CUBRID HA feature is used. Failover is supported for its node.
- **replica** : CUBRID HA feature is used. Failover is not supported for its node.

The **ha_mode** parameter can be re-configured in the **[@<database>]** section; however, only **off** can be entered in the case. An error will be outputted if a value other than **off** is entered in the **[@<database>]** section.

If **ha_mode** is **on**, the CUBRID HA values are configured by reading **cubrid_ha.conf**.

This parameter cannot be modified dynamically. To modify the value of this parameter, you must restart the corresponding node.

log_max_archives

This parameter configures the minimum number of archive log files to be archived when the value of **media_failure_support** is yes. The minimum value is 0, and the default is **INT_MAX**. The performance of the parameter is affected by **force_remove_log_archives**.

The existing archive log files to which the activated transaction refers or the archive log files of the master node not reflected to the slave node in HA environment will not be deleted. For more information, see the following **force_remove_log_archives**.

For more information about **log_max_archives**, see Logging-related Parameters.

force_remove_log_archives

It is recommended to always maintain the archive logs needed to perform the HA related process by setting a value for **force_remove_log_archives** to no in order to implement the HA environment by setting a value for **ha_mode** to on.

If you set for **force_remove_log_archives** to yes, the archive log files which will be used in the HA related process can be deleted, and this may lead to an inconsistency between replicated databases. If you want to maintain free disk space even with this risk, set for **force_remove_log_archives** to yes.

For more information about **force_remove_log_archives**, see Logging-related Parameters.

max_clients

A parameter used to specify the maximum number of clients to be connected to a database server simultaneously. The default is **50**.

Because the replication log copy process and the replication log reflection process are started by default if CUBRID HA feature is used, you must configure the value to twice the number of all nodes in the CUBRID HA group, except the corresponding node. Furthermore, you must consider the case in which a client that was connected to another node at the time of failover attempts to connect to that node.

Example

The following is an example of configuring **cubrid.conf**:

```
max_clients=200

ha_mode=on
log_max_archives=100
```

cubrid_ha.conf

The **cubrid_ha.conf** file that has generation information on CUBRID HA is located in the **$CUBRID/conf** directory.

ha_node_list

A parameter used to specify the group name to be used in the CUBRID HA group and the host name of member nodes in which failover is supported. You can separate names of group and their member nodes by using an "at" mark (@). The default is **localhost@localhost**.

The followings must be registered in **/etc/hosts**: 1) the host names of member nodes specified in this parameter, 2) the host name of a node to be configured.

A node in which the **ha_mode** value is set to **on** must be specified in **ha_node_list**. The value of the **ha_node_list** of all nodes in the CUBRID HA group must be identical. When a failover occurs, a master node is determined in the order specified in the parameter.

This parameter can be modified dynamically. If you modify the value of this parameter, you must execute cubrid heartbeat reload to apply the changes.

ha_replica_list

A parameter used to specify the group name to be used in the CUBRID HA group and the host name of member nodes in which failover is not supported. You can separate names of group and their member nodes by using an "at" mark (@). The default is **NULL**.

The group name must be identical to the name specified in **ha_node_list**.

The followings must be registered in **/etc/hosts**: 1) the host names of member nodes specified in this parameter, 2) the host name of a node to be configured.

A node in which the **ha_mode** value is set to **replica** must be specified in **ha_replica_list**. The **ha_node_list** values of all nodes in the CUBRID HA group must be identical.

This parameter can be modified dynamically. If you modify the value of this parameter, you must execute cubrid heartbeat reload to apply the changes.

ha_port_id

A parameter used to specify the UDP port number; the UDP port is used to detect failure when exchanging heartbeat messages. The default is **59901**.

If a firewall exists in the service environment, the firewall must be configured to allow the configured port to pass through it.

ha_ping_hosts

A parameter used to specify the host which verifies whether or not a failover occurs due to unstable network when a failover has started in a slave node. The default is **NULL**.

Configuring this parameter can prevent failovers from occurring due to network instability. You can specify multiple hosts by using a colon (:).

ha_copy_sync_mode

A parameter used to specify the mode of storing the transaction log copy. The default is **SYNC**.

The value can be one of the followings: **SYNC**, **SEMISYNC**, or **ASYNC**. The number of values must be the same as the number of nodes specified in **ha_node_list**. They must be ordered by the specified value. You can specify multiple nodes by using a colon (:)

For more information, see Multiplexing Logs.

ha_copy_log_base

A parameter used to specify the location of storing the transaction log copy. The default is **$CUBRID_DATABASES**.

For more information, see Multiplexing Logs.

ha_db_list

A parameter used to specify the name of the database that will run in CUBRID HA mode. The default is **NULL**. You can specify multiple databases by using a colon (:).

ha_apply_max_mem_size

A parameter to used to specify the value of maximum memory that the replication log reflection process of CUBRID HA can use. The default is **500**.

Example

The following is an example of configuring **cubrid_ha.conf**:

```
[common]
ha_node_list=foo@apple:banana
ha_db_list=testdb
ha_copy_sync_mode=sync:semisync
ha_apply_max_mem_size=1000
```

cubrid_broker.conf

The **cubrid_broker.conf** file that has general information on configuring CUBRID broker is located in the **$CUBRID/conf** directory. This section explains the parameters of **cubrid_broker.conf** that are used by CUBRID HA.

ACCESS_MODE

A parameter used to specify the mode of a broker. The default is **RW**.

Its value can be one of the followings: **RW** (Read Write), **RO** (Read Only), **SO** (Slave Only), or **PHRO** (Preferred Host Read Only). For more information, see Broker mode.

PREFERRED_HOSTS

A parameter used in order to configure the broker mode to Preferred Host Read Only. The default is **NULL**.

You can specify multiple nodes by using a colon (:).

Example

The following is an example of configuring **cubrid_broker.conf**.

```
[%testdb_broker]
SERVICE                 =ON
BROKER_PORT             =33000
MIN_NUM_APPL_SERVER     =5
MAX_NUM_APPL_SERVER     =40
APPL_SERVER_SHM_ID      =33000
LOG_DIR                 =log/broker/sql_log
ERROR_LOG_DIR           =log/broker/error_log
SQL_LOG                 =ON
TIME_TO_KILL            =120
SESSION_TIMEOUT         =300
KEEP_CONNECTION         =AUTO
ACCESS_MODE             =RW
PREFERRED_HOSTS         =nodeA:nodeB:nodeC
```

databases.txt

The databases.txt file that has information on servers to be connected by a broker and their order is located in the **$CUBRID_DATABASES** directory; the information can be configured by using **db_hosts**. You can specify multiple nodes by using a colon (:).

The following is an example of configuring databases.txt:

```
#db-name     vol-path          db-host      log-path
tdb01        /home/cubrid/DB/tdb01 masterdb.cub:slavedb.cub    /home/cubrid/DB/tdb01/log
tdb02        /home/cubrid/DB/tdb02 masterdb.cub:slavedb.cub    /home/cubrid/DB/tdb02/log
```

JDBC Configuration

To use the CUBRID HA feature in JDBC, you must specify the connection information of another broker to be connected when a failure occurs in the connection URL. The attributes specified for CUBRID HA include **althosts** which is the information of more than one broker node to be connected when a failure occurs and **rctime** which is the reconnection interval after the recovery of a failed broker. For more information, see Configuring Connection.

The following is an example of configuring JDBC:

```
Connection connection =
DriverManager.getConnection("jdbc:cubrid:primary_broker:33000:testdb01:::?charset=utf-
8&althosts=secondary_broker:45011&rctime=30", "dba", "");
```

CCI Configuration

To use the CUBRID HA feature in CCI, you must connect it to the broker by using cci_connect_with_url, which is used to specify the connection information of the failover broker in the connection URL. The attributes specified for CUBRID HA include **althosts** which is the information of more than one broker node to be connected when a failure occurs and **rctime** which is the reconnection interval after the recovery of a failed broker.

The following is an example of configuring CCI.

```
con = cci_connect_with_url
("cci:cubrid:primary_broker:33000:tdb01:::?althosts=secondary_broker:45011&rctime=30",
"dba", NULL);
if (con < 0)
{
      printf ("cannot connect to database\n");
      return 1;
}
```

Running and Monitoring

Utilities of cubrid heartbeat

start

This utility is used to start all components of CUBRID HA in the node (database server process, replication log copy process, replication log reflection process).

Note that a master node or a slave node is determined based on the execution order of **cubrid heartbeat start**.

How to execute the command is as shown below.

```
$ cubrid heartbeat start
$
```

cubrid server start only starts cub_server process of the database, regardless of HA mode configuration. If you want to start all HA related processes, you can execute **cubrid heartbeat start**.

stop

This utility is used to stop all components of CUBRID HA in the node (database server process, replication log copy process, replication log reflection process). The node that executes this command stops and a failover occurs to the next slave node according to the CUBRID HA configuration.

How to use this utility is as shown below.

```
$ cubrid heartbeat stop
$
```

cubrid server stop only starts cub_server process of the database, regardless of HA mode configuration. The database does not restart, and failover does not occur. If you want to stop all HA related processes, you can execute **cubrid heartbeat stop**.

reload

This utility is used to retrieve the CUBRID HA information again, and it starts or stops the CUBRID HA components according to new CUBRID HA configuration. You can modify the information of **ha_node_list** and **ha_replica_list**. If an error occurs during the command execution, the node will stop.

How to use this utility is as shown below.

```
$ cubrid heartbeat reload
$
```

deact

This utility is used to exclude the node from the CUBRID HA group. A node in which **deact** is executed will be excluded from the CUBRID HA group and the CUBRID HA components will stop. The status of this node is outputted as **unknown** when you verify it by using **cubrid heartbeat status**. You can include the node to the CUBRID HA group back by executing **act**.

It is recommended that this command be used only when it is unavoidable.

How to use this utility is as shown below.

```
$ cubrid heartbeat deact
$
```

act

This utility is used to includes nodes back in the CUBRID HA group, and it starts the CUBRID HA components.

It is recommended that this command only be used when it is unavoidable.

How to use this utility is as shown below.

```
$ cubrid heartbeat act
$
```

status

This utility is used to output the information of CUBRID HA group and CUBRID HA components.

How to use this utility is as shown below.

```
$ cubrid heartbeat status
@ cubrid heartbeat list

 HA-Node Info (current slaved.cub, state slave)
   Node slavedb.cub (priority 2, state slave)
   Node masterdb.cub (priority 1, state master)


 HA-Process Info (master 2143, state slave)
   Applylogdb tdb01@localhost:/home/cubrid/DB/tdb01_slavedb.cub (pid 2510, state
registered)
   Copylogdb tdb01@masterdb.cub:/home/cubrid/DB/tdb01_masterdb.cub (pid 2505, state
registered)
   Server tdb01 (pid 2393, state registered)

$
```

Utilities of cubrid service

If you register heartbeat to CUBRID service, you can use the utilities of **cubrid service** to start, stop or check all the related processes at once. The processes specified by **service** parameter in [**service**] section in **cubrid.conf** file are registered to CUBRID service. If this parameter includes **heartbeat**, you can start/stop all the service processes and the HA related processes by using **cubrid service start**/**stop** command.

How to configure **cubrid.conf** file is shown below.

```
# cubrid.conf
...
[service]
...
service=broker,heartbeat
...
[common]
...
ha_mode=on
```

cubrid applyinfo

Description

This utility is used to monitor the replication status of CUBRID HA.

Syntax

```
cubrid applyinfo [option] <database-name>
```

- *database-name* : Specifies the name of a server to monitor. A node name is not included.

Option

Option	Default	Description
-r	none	Configures the name of a target node in which transaction logs are copied. Using

		this option will output the information of active logs (Active Info.) of a target node.
-a		Outputs the information of replication reflection of a node executing cubrid applyinfo. The **-L** option is required to use this option.
-L	none	Configures the location of transaction logs copied from the other node. Using this option will output the information of transaction logs copied (Copied Active Info.) from the other node.
-p	0	Outputs the information of a specific page in the copied logs. This is available only when the **-L** option is enabled.
-v		Outputs detailed information.

Example

```
$ cubrid applyinfo -L /home/cubrid/DB/tdb01_masterdb.cub -r master_node_name -a tdb01

 *** Applied Info. ***
Committed page                 : 1913 | 2904
Insert count                   : 645
Update count                   : 0
Delete count                   : 0
Schema count                   : 60
Commit count                   : 15
Fail count                     : 0

 *** Copied Active Info. ***
DB name                        : tdb01
DB creation time               : 11:28:00.000 AM 12/17/2010  (1292552880)
EOF LSA                        : 1913 | 2976
Append LSA                     : 1913 | 2976
HA server state                : active

 ***  Active Info. ***
DB name                        : tdb01
DB creation time               : 11:28:00.000 AM 12/17/2010  (1292552880)
EOF LSA                        : 1913 | 2976
Append LSA                     : 1913 | 2976
HA server state                : active
$
```

- Applied Info.
- Committed page : The information of committed pageid and offset of a transaction reflected last through replication log reflection process. This information is internally used; which means that replication reflection will be delayed if a big difference exists between the EOF LSA value of "Copied Active Info." and this value.
- Insert Count : The number of Insert queries reflected through replication log reflection process.
- Update Count : The number of Update queries reflected through replication log reflection process.
- Delete Count : The number of Delete queries reflected through replication log reflection process.
- Schema Count : The number of DDL statements reflected through replication log reflection process.
- Commit Count : The number of commits reflected through replication log reflection process.
- Fail Count : The number of DML and DDL statements in which log reflection through replication log reflection process fails.
- Copied Active Info.
- DB name : Database name of a database server process to which replication log copy process copies replication logs
- DB creation time : The creation time of a database copied through replication log copy process
- EOF LSA : The last information of pageid and offset of a database server process replication log copied through replication log copy process. There will be a delay in replication log copy process as much as difference with the EOF LSA value of "Active Info." and with the Append LSA value of "Copied Active Info."
- Append LSA : The last information of pageid and offset of a log received from the database server process through replication log copy process. This value can be less than or equal to EOF LSA. There will be a delay in replication log copy process as much as difference between the EOF LSA value of "Copied Active Info." and this value.
- HA server state : Status of a database server process which replication log copy process receives replication logs from. For more information on status, see Server.

- Active Info.
- DB name : Database name of a database server process of a node that is configured in the **-r** option.
- DB creation time : Database creation time of a node that is configured in the **-r** option.
- EOF LSA : The last information of pageid and offset of a database server process replication log of a node that is configured in the **-r** option. There will be a delay in replication log copy process as much as difference between the EOF LSA value of "Copied Active Info." and this value.
- Append LSA : The last information of pageid and offset of a replication log which is written in a database server process of a node that is configured in the **-r** option.
- HA server state : Status of a database server process of a node that is configured in the **-r** option.

cubrid changemode

Description

This utility is used to check and change the server status of CUBRID HA.

Syntax

```
cubrid changemode [option] <database-name>
```

- *database-name* : Specifies the name of a server to monitor and the node name; separate them by using @.

Option

Option	Default	Description
-m	none	Changes the server status. You can enter one of the followings: **standby**, **maintenance**, or **active**.
-f		Configures whether or not to forcibly change the server status. This option must be configured if you want to change the server status from to-be-active to active. If it is not configured, the status will not be changed to active. Forcibly change may cause replication inconsistency; so it is not recommended.

Status Changeable Map

This table shows changeable modes depending on current status. However, replication inconsistency may occur if the status of the current server is changed from to-be-active to active. Therefore, it is recommended that only a user who is familiar with this condition uses this option.

		Future Status		
		active	standby	maintenance
Current Status	standby	X	O	O
	to-be-standby	X	X	X
	active	O	X	X
	to-be-active	O	X	X
	maintenance	X	O	O

Note When the server status is to-be-active, forcibly change may cause replication inconsistency. It is not recommended if you are not skilled enough.

Example

The following is an example of changing the testdb01 server status in a localhost node to maintenance.

```
$ cubrid changemode -m maintenance tdb01@localhost
```

```
The server `tdb01@localhost''s current HA running mode is maintenance.
$
```

The following is an example of retrieving the testdb01 server status in the a localhost node.

```
$ cubrid changemode tdb01@localhost
The server `tdb01@localhost''s current HA running mode is active.
$
```

Monitoring CUBRID Manager HA

CUBRID Manager is a dedicated CUBRID database management tool that provides the CUBRID database management and query features in a GUI environment. CUBRID Manager provides the HA dashboard, which shows the relationship diagram for the CUBRID HA group and server status. For more information, see CUBRID Manager manual.

Configuration

Overview

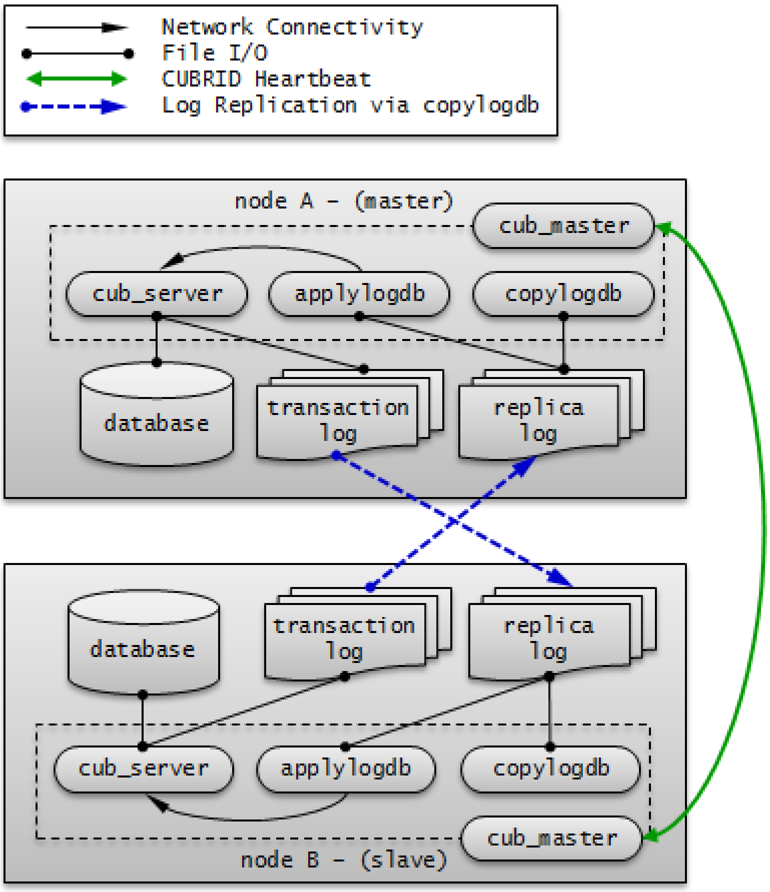
There are four possible structures for CUBRID HA: HA basic structure, multiple-slave node structure, load balancing structure, and multiple-standby server structure. In the table below, M stands for a master node, S for a slave node, and R for a replica node.

Structure	Node structure (M:S:R)	Characteristic
Basic Structure	1:1:0	The most basic structure of CUBRID HA consists of one master node and one slave node and provides availability which is a unique feature of CUBRID HA.
Multiple-Slave Node Structure	1:N:0	This is a structure in which availability is increased by several slave nodes. However, note that there may be a situation in which data is inconsistent in the CUBRID HA group when multiple failures occur.
Load Balancing Structure	1:1:N	Several replica nodes are added in the basic structure. Read service load can be distributed, and the HA load is reduced, comparing to a multiple-slave node structure. Note that replica nodes do not failover.
Multiple-Standby Server Structure	1:1:0	Basically, this structure is the same as the basic structure. However, several slave nodes are installed on a single physical server.

Basic Structure

The most basic structure of CUBRID HA consists of one master node and one slave node.

As the basic structure focusing on the unique feature of CUBRID HA that provides an uninterrupted service during a failover, this structure requires a small amount of resources on a small scale service. Because there are only one master node and one slave node for service, the read load may be concentrated to these two nodes during read-heavy operations.



An Example of Node Configuration

You can configure each node in the basic structure of HA as shown below:

- **nodeA** (master node)
- Configure the **ha_mode** of the **cubrid.conf** file to **on**.

```
ha_mode=on
```

- The following is an example of configuring **cubrid_ha.conf**:

```
ha_port_id=12345
ha_node_list=cubrid@nodeA:nodeB
ha_db_list=testdb1:testdb2
```

- **nodeB** (slave node): Configure this node in the same manner as nodeA.

For the **databases.txt** file of a broker node, it is necessary to configure the list of hosts configured as HA in **db-host** according to their priority. The following is an example of the **databases.txt** file:

```
#db-name     vol-path                   db-host        log-path
testdb1      /home/cubrid/DB/testdb1    nodeA:nodeB    /home/cubrid/DB/testdb1/log
```

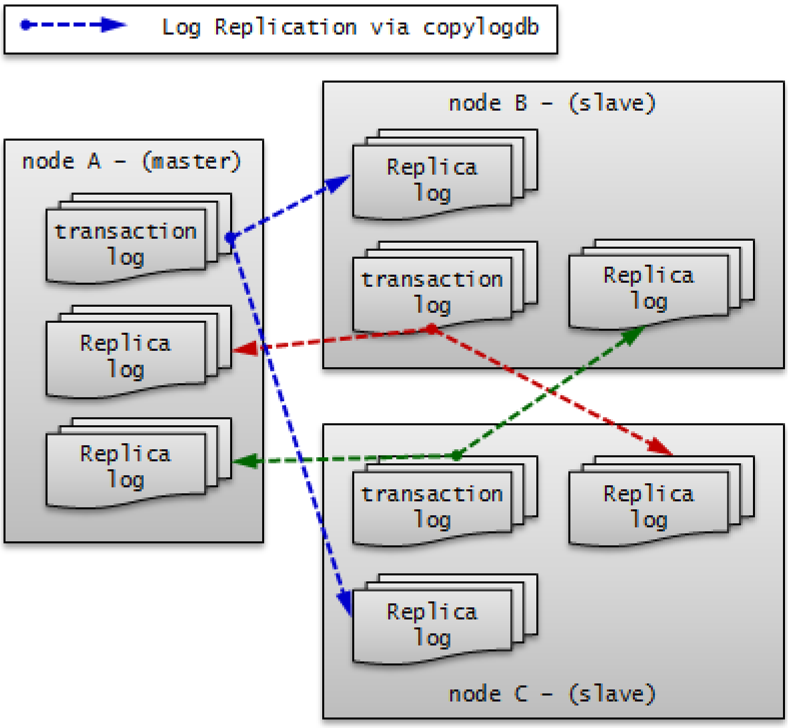
Multiple-Slave Node Structure

In multiple-slave node structure, there is one master node and several slave nodes to improve the service availability of CUBRID.

Because replication log copy process and replication log reflection process are running at all nodes in the CUBRID HA group, a load of copying replication log occurs. Therefore, all nodes in the CUBRID HA group have high network and disk usage.

Because there are many nodes with HA enabled, read and write services never fail as long as a single node is alive.

In the multiple-slave node structure, the node becoming a master node when failover occurs is determined by the order specified in **ha_node_list**. If the value of **ha_node_list** is node1:node2:node3 and the master node is node1, node2 will become a new master node when node1 fails.



An Example of Node Configuration

You can configure each node in the basic structure of HA as shown below:

- **node A** (master node)
- Configure the **ha_mode** of the **cubrid.conf** file to **on**.

```
ha_mode=on
```

- The following is an example of configuring **cubrid_ha.conf**:

```
ha_port_id=12345
ha_node_list=cubrid@nodeA:nodeB:nodeC
ha_db_list=testdb1:testdb2
```

- **node B** (slave node): Configure this node in the same manner as nodeA.
- **node C** (slave node): Configure this node in the same manner as nodeA.

For the **databases.txt** file of a broker node, it is necessary to configure the list of hosts configured as HA in **db-host** according to their priority. The following is an example of the **databases.txt** file:

```
#db-name    vol-path               db-host           log-path
testdb1     /home/cubrid/DB/testdb1   nodeA:nodeB:nodeC   /home/cubrid/DB/testdb1/log
```

Caution

The data in the CUBRID HA group may lose integrity when there are multiple failures in this structure.

- In a situation where a failover occurs in the first slave node while replication in the second slave node is being delayed due to restart
- In a situation where a failover re-occurs before replication reflection of a new master node is not complete due to frequent failover

In addition, if the mode of replication log copy process is ASYNC, the data in the CUBRID HA group may lose integrity.

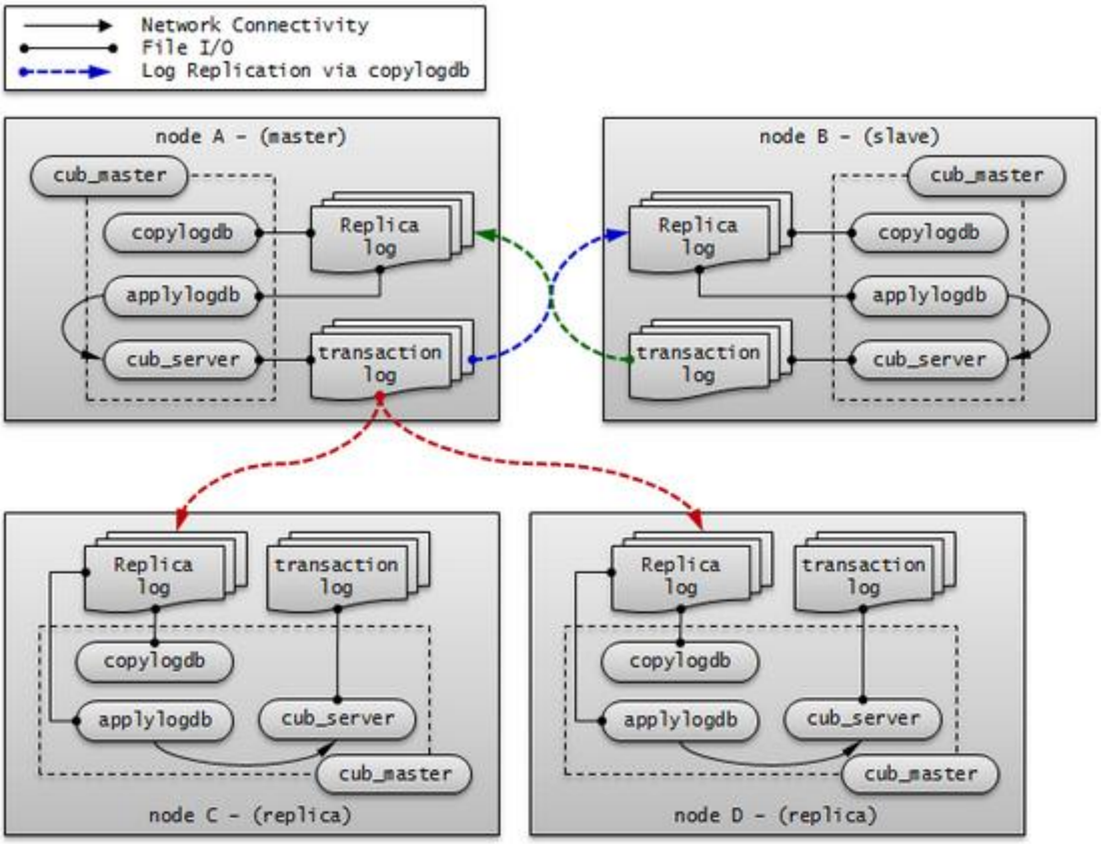
If the data in the CUBRID HA group loses integrity for any of the reasons above, you can fix it by using Rebuilding Replications.

Load Balancing Structure

The load balancing structure increases the availability of the CUBRID servie by placing several nodes in the HA configuration (one master node and one slave node) and distributes read-load.

Because the replica nodes receive replication logs from the nodes in the HA configuration and maintain the same data, and because the nodes in the HA configuration do not receive replication logs from the replica nodes, its network and disk usage rate is lower than that of the multiple-slave structure.

Because replica nodes are not included in the HA structure, they provide read service without failover, even when all other nodes in the HA structure fail.



An Example of Node Configuration

You can configure each node in the basic structure of HA as shown below:

- **node A** (master node)
- Configure the **ha_mode** of the **cubrid.conf** file to **on**.

```
ha_mode=on
```

- The following is an example of configuring **cubrid_ha.conf**:

```
ha_port_id=12345
ha_node_list=cubrid@nodeA:nodeB
ha_replica_list=cubrid@nodeC:nodeD
ha_db_list=testdb1:testdb2
```

- **node B** (slave node): Configure this node in the same manner as nodeA.
- **node C** (replica node)
- Configure the **ha_mode** of the **cubrid.conf** file to **replica**.

```
ha_mode=replica
```

- You can configure the **cubrid_ha.conf** file in the same manner as nodeA.
- **node D** (replica node): Configure this node in the same manner as nodeC.

You must enter the list of DB server hosts in the order so that each broker can be connected appropriate HA or load balancing server in the **databases.txt** file of a broker node. The following is an example of the **databases.txt** file:

```
#db-name    vol-path                  db-host      log-path
testdb1     /home/cubrid/DB/testdb1   nodeA:nodeB  /home/cubrid/DB/testdb1/log
testdb2     /home/cubrid/DB/testdb2   nodeC:nodeD  /home/cubrid/DB/testdb2/log
```

Caution

The data in the CUBRID HA group may lose integrity when there are multiple failures in this structure.

- In a situation where a failover occurs in the first slave node while replication in the second slave node is being delayed due to restart
- In a situation where a failover re-occurs before replication reflection of a new master node is not complete due to frequent failover

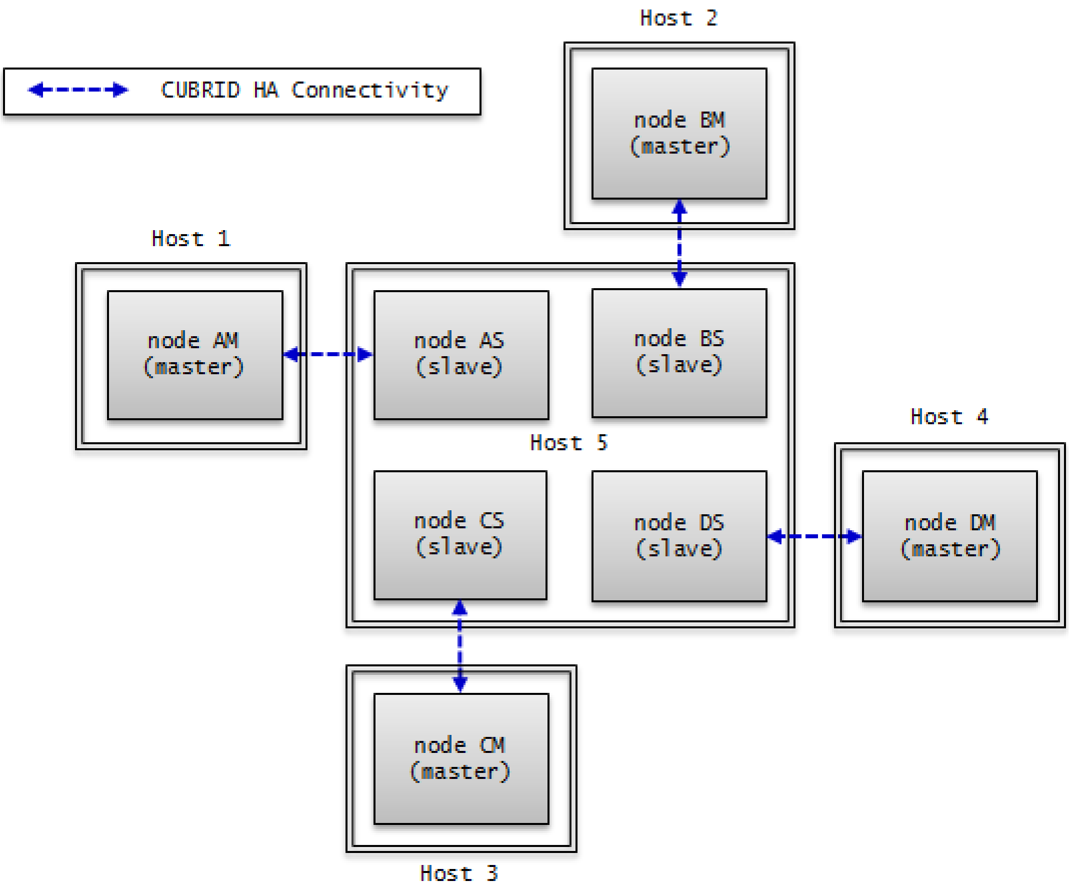
In addition, if the mode of replication log copy process is ASYNC, the data in the CUBRID HA group may lose integrity.

If the data in the CUBRID HA group loses integrity for any of the reasons above, you can fix it by using Rebuilding Replications.

Multiple-Standby Server Structure

Although its node structure has a single master node and a single slave node, many slave nodes from different services are physically configured in a single server.

This structure is for very small services in which the read load of slave nodes are light. It is strictly for the availability of the CUBRID service. For this reason, when a master node that failed after a failover has been restored, the load must be moved back to the original master node to minimize the load of the server with multiple-slave nodes.



An Example of Node Configuration

You can configure each node in the basic structure of HA as shown below:

- **node AM**, **node AS**: Configure node AM and node AS in the same manner.
- Configure the **ha_mode** of the **cubrid.conf** file to **on**.

```
ha_mode=on
```

- The following is an example of configuring **cubrid_ha.conf**:

```
ha_port_id=10000
ha_node_list=cubridA@Host1:Host5
ha_db_list=testdbA1:testdbA2
```

- **node BM**, **node BS**: Configure node BM and node BS in the same manner.
- Configure the **ha_mode** of the **cubrid.conf** file to **on**.

```
ha_mode=on
```

- The following is an example of configuring **cubrid_ha.conf**:

```
ha_port_id=10001
ha_node_list=cubridB@Host2:Host5
ha_db_list=testdbB1:testdbB2
```

- **node CM**, **node CS**: Configure node CM and node CS in the same manner.
- Configure the **ha_mode** of the **cubrid.conf** file to **on**.

```
ha_mode=on
```

- The following is an example of configuring **cubrid_ha.conf**:

```
ha_port_id=10002
```

```
ha_node_list=cubridC@Host3:Host5
```

```
ha_db_list=testdbC1:testdbC2
```

- **node DM**, **node DS**: Configure node DM and node DS in the same manner.
- Configure the **ha_mode** of the **cubrid.conf** file to **on**.

```
ha_mode=on
```

- The following is an example of configuring **cubrid_ha.conf**:

```
ha_port_id=10003
```

```
ha_node_list=cubridD@Host4:Host5
```

```
ha_db_list=testdbD1:testdbD2
```

Constraints

Supported Platform

Currently, CUBRID HA feature is supported only by Linux. All nodes within a CUBRID HA group must be configured with the same platform.

Table Primary Key

CUBRID HA synchronizes data among nodes with the following method (as known as transaction log shipping): First, it replicates the primary key-based replication logs generated from the server of a master node to a slave node. Second, it reflects the replication logs.

If data of the specific table within a CUBRID HA group is not synchronized, you should check whether the appropriate primary key has specified for the table.

Table Trigger and Java Stored Procedure

Using triggers and java stored procedures in CUBRID HA can cause duplicate execution because triggers and java stored procedures executed in a master node will be executed in a slave node again. This may cause data inconsistency among nodes within a CUBRID HA group.

It is not recommended to use triggers and java stored procedures in CUBRID HA.

Method and CUBRID Manager

CUBRID HA synchronizes data among nodes within a CUBRID HA group based on replication logs. Therefore, using method that does not generate replication logs or configuring **NOT NULL** through CUBRID Manager may cause data inconsistency among nodes within a CUBRID HA group.

Stand-Alone Mode

The replication logs are not generated as for tasks performed in stand-alone mode. For this reason, data inconsistency among nodes in a CUBRID HA group may occur when performing tasks in stand-alone mode.

Serial Cache

To enhance performance, a serial cache does not access Heap and does not generate replication logs when retrieving or updating serial information. Therefore, if you use a serial cache, the current values of serial caches will be inconsistent among the nodes in a CUBRID HA group.

cubrid backupdb -r

This command is used to back up a specified database. If the **–r** option is used, logs that are not required for recovery will be deleted. This deletion may result in data inconsistency among nodes in a HA group. Therefore, you must not use the **-r** option.

INCR/DECR functions

The click counter functions **INCR/DECR** should not be used in slave nodes of HA. If you use these functions in slave nodes, data inconsistency may occur between slave nodes and master nodes.

Error Messages

Replication Log Copy Process

The error messages from the replication log copy process are stored in **$CUBRID/log/***db-name***@***remote-node-name***_copylogdb.err**. The severity of error messages found in the replication log copy process are as follows: fatal, error, and notification. The default severity is error. Therefore, to store notification error messages, it is necessary to change the value of **error_log_level** in **cubrid.conf**. For more information, see Error Message-Related Parameters.

Initialization Error Message

The error messages that may be stored in the initialization stage of replication log copy process are as follows:

Error Code	Error Name	Severity	Description	Solution
10	ER_IO_MOUNT_FAIL	error	Opening a replication log file has failed.	Check if there is a replication log. For the location of replicated logs, see Default Environment Configuration.
78	ER_LOG_READ	fatal	Reading a replication log has failed.	Check the replication log by using the cubrid applyinfo utility.
81	ER_LOG_PAGE_CORRUPTED	fatal	A replication log page error, in which the replication log copy process has been copied from the connected database server process.	Check the error log of the database server process to which the replication log copy process is connected. This error log can be found in $CUBRID/log/server.
1039	ER_HA_LW_STARTED	error	The replication log copy process has been successfully initialized and started.	No action is required, because this error message has been stored to display the starting information of the replication log copy process. Error messages that have been displayed after the start of the replication log copy process and before this error message is displayed can be ignored, as they can be displayed in normal conditions.

Replication Log Request and Reception Error Messages

The replication log copy process requests a replication log from the connected database server, and receives the corresponding replication log. The error messages that may occur during this process are as follows:

Error Code	Error Name	Severity	Description	Solution
89	ER_LOG_DOESNT_CORRESPOND_TO_DATABASE	error	The previously	Check the information

			replicated log and the log to be replicated do not match.	of the database server/host to which the replication log copy process is connected. If you need to change the database server/host information, reinitialize it by deleting the existing replication log, and then restarting.
186	ER_NET_SERVER_DATA_RECEIVE	error	Incorrect information has been received from the database server to which the replication log copy process is connected.	This is recovered internally.
199	ER_NET_SERVER_CRASHED	error	The connection to the database server has been terminated.	This is recovered internally.

Replication Log Writing Error Messages

The replication log copy process copies the replication log (ha_copy_base) that was received from the connected database server process to the location specified in **cubrid_ha.conf**. The error messages that may occur during this process are as follows:

Error Code	Error Name	Severity	Description	Solution
10	ER_IO_MOUNT_FAIL	error	Opening a replication log file has failed.	Check if the replication log exists.
79	ER_LOG_WRITE	fatal	Writing a replication log has failed.	This is recovered internally.
80	ER_IO_WRITE_OUT_OF_SPACE	fatal	Writing a replication log has failed due to insufficient file system space.	Check if there is sufficient space left in the disk partition.

Replication Log Archive Error Messages

The replication log copy process periodically archives the replication log that has been received from the connected database server process. The error messages that may occur during this process are as follows:

Error Code	Error Name	Severity	Description	Solution
78	ER_LOG_READ	fatal	Reading a replication log has failed during archiving.	Check the replication log by using the cubrid applyinfo utility.
79	ER_LOG_WRITE	fatal	Writing the archive log has failed.	This is recovered internally.
81	ER_LOG_PAGE_CORRUPTED	fatal	A replication log error has been found during archiving.	Check the replication log by using the cubrid applyinfo utility.
98	ER_LOG_CREATE_LOGARCHIVE_FAIL	fatal	Creating the archive log file has failed.	Check if there is sufficient space left in the disk partition.
974	ER_LOG_ARCHIVE_CREATED	notification	Archive log file information	No action is required, because this error message is recorded for the log information of the newly created archive.

Stop and Restart Error Message

The error messages that may occur at the beginning and the end of the replication log copy process are as follows:

Error Code	Error Name	Severity	Description	Solution
1037	ER_HA_LW_STOPPED_BY_SIGNAL	error	The copylogdb process has been terminated by a specific signal.	This is recovered internally.

Replication Log Reflection Process

The error messages from the replication log reflection process are stored in **$CUBRID/log/***db-name@local-node-name***_applylogdb_***db-name_remote-node-name***.err**. The severity of error messages found in the replication log reflection process can be as follows: fatal, error, and notification. The default severity is error. Therefore, to store notification error messages, it is necessary to change the value of **error_log_level** in **cubrid.conf**. For more information, see Error Message-Related Parameters.

Initialization Error Message

The error messages that may be stored in the initialization stage of the replication log reflection process are as follows:

Error Code	Error Name	Severity	Description	Solution
10	ER_IO_MOUNT_FAIL	error	An applylogdb that is trying to reflect the same replica log is	Check if there is a applylogdb process that is trying to reflect the same replication log.

		already running.	
1038	ER_HA_LA_STARTED error	Starts normally once the initialization of applylogdb is successful.	No action is required, because this error message has been stored to display the starting information of the replication log reflection process.

Log Analysis Error Message

The replication log reflection process reads, analyzes, and reflects the replication logs that have been copied by the replication log copy process. The error messages that may occur during a replication log analysis are as follows:

Error Code	Error Name	Severity	Description	Solution
13	ER_IO_READ	error	Reading the log page to be reflected has failed.	Check the replication log by using the cubrid applyinfo utility.
17	ER_PB_BAD_PAGEID	fatal	Trying to read a log page that does not exist in the replication log	Check the replication log by using the cubrid applyinfo utility.
81	ER_LOG_PAGE_CORRUPTED	fatal	There is an inconsistency between an old log under replication reflection and the current log, or there is a replication log record error.	Check the replication log by using the cubrid applyinfo utility.
82	ER_LOG_MOUNT_FAIL	error	No replication log file	Check if there is a replication log. Check the replication log by using the cubrid applyinfo utility.
97	ER_LOG_NOTIN_ARCHIVE	error	The log page does not exist in the replication log.	Check the replication log by using the cubrid applyinfo utility.
897	ER_IO_LZO_DECOMPRESS_FAIL	error	Decompressing the log record has failed.	Check the replication log by using the

				cubrid applyinfo utility.
1028	ER_HA_LA_UNEXPECTED_EOF_IN_ARCHIVE_LOG	error	An incorrect log record exists in the archive log.	Check the replication log by using the cubrid applyinfo utility.
1029	ER_HA_LA_INVALID_REPL_LOG_PAGEID_OFFSET	error	An incorrect log record has been included.	Check the replication log by using the cubrid applyinfo utility.
1030	ER_HA_LA_INVALID_REPL_LOG_RECORD	error	A log record header error	Check the replication log by using the cubrid applyinfo utility.

Replication Log Reflection Error Message

The replication log reflection process reads, analyzes, and reflects the replication logs that have been copied by the replication log copy process. The error messages that may occur when reflecting a replication log analysis are as follows:

Error Code	Error Name	Severity	Description	Solution
72	ER_LK_UNILATERALLY_ABORTED	error	Replication reflection has been failed due to deadlock, etc.	This is recovered internally.
111	ER_TM_SERVER_DOWN_UNILATERALLY_ABORTED	error	Replication reflection is failed because the database server process for replication reflection has been terminated, or its mode has been changed.	This is recovered internally.
191	ER_NET_CANT_CONNECT_SERVER	error	The connection to the database server process for replication reflection has been	This is recovered internally.

			terminated.	
195	ER_NET_SERVER_COMM_ERROR	error	The connection to the database server process for replication reflection has been terminated.	This is recovered internally.
224	ER_OBJ_NO_CONNECT	error	The connection to the database server process for replication reflection has been terminated.	This is recovered internally.
1027	ER_HA_LA_FAILED_TO_CHANGE_STATE	error	The status change of replication reflection has been failed.	This is recovered internally.
1031	ER_HA_LA_FAILED_TO_APPLY_SCHEMA	error	SCHEMA replication reflection has been failed.	Check the consistency of the replica. If it is inconsistent, reconfigure the HA replication.
1032	ER_HA_LA_FAILED_TO_APPLY_INSERT	error	INSERT replication reflection has been failed.	Check the consistency of the replica. If it is inconsistent, reconfigure the HA replication.
1033	ER_HA_LA_FAILED_TO_APPLY_UPDATE	error	UPDATE replication reflection has been failed.	Check the consistency of the replica. If it is inconsistent, reconfigure the HA replication.
1034	ER_HA_LA_FAILED_TO_APPLY_DELETE	error	DELETE replication reflection has been failed.	Check the consistency of the replica. If it is

				inconsistent, reconfigure the HA replication.
1040	ER_HA_GENERIC_ERROR	notification	Change the last record of the archive log or replication reflection status.	No action is required, because this error message is stored to provide general information.

Stop and Restart Error Message

The error messages that may occur at the beginning and the end of the replication log reflection process are as follows:

Error Code	Error Name	Severity	Description	Solution
1035	ER_HA_LA_EXCEED_MAX_MEM_SIZE	error	The replication log reflection process has been restarted due to reaching the maximum memory size limit.	This is recovered internally.
1036	ER_HA_LA_STOPPED_BY_SIGNAL	error	The replication log reflection process has been terminated by a specified signal.	This is recovered internally.

Operation Scenario

Rebuilding Replications

Replication rebuilding is required in CUBRID HA when data in the CUBRID HA group is inconsistent because of multiple failures in multiple-slave node structure, or because of a generic error. Rebuliding replications in CUBRID HA is done using a script.

For rebuilding replications to happen, the following environment must be the same in the slave node and in the master node.

- CUBRID version
- Environmental variable ($CUBRID, $CUBRID_DATABASES, $LD_LIBRARY_PATH, $PATH)
- Database volume, log and replication log path

ha_make_slavedb.sh script

To rebuild replications, use the **ha_make_slavedb.sh**. This script is located in **$CUBRID/share/script/ha**. Before rebuilding replications, the following items must be configured for the environment of the user.

- **master_host**: The host name of the master node during replication rebuilding. It should be registered in **/etc/hosts**.
- **db_name**: Specifies the name of the database to be replicated and rebuilt.
- **repl_log_home**: Specifies the home directory of the replication log of the master node. It is usually the same as **$CUBRID_DATABASES**.

The following are optional items:

- **dba_password**: If the CUBRID dba account is password protected, sets the password.
- **backup_dest_path**: When executing **backupdb** at the master node, configures the path in which the backup volume will be created.
- **backup_option**: Configures necessary options when executing **backupdb** at the master node.

- **restore_option**: Configures necessary options when executing a **restore** at the slave node in which the replication will be rebuilt.
- **scp_option**: An option that is used to configure the **scp** option to copy the backup volume of the master node to the slave node. Its default value is **16M** so it will not cause network overload in the master node.

Once the script has been configured, execute the **ha_make_slavedb.sh** script at the slave node in which a replication will be rebuilt. When the script is executed, replication rebuilding occurs in a number of phases. To move to the next stage, the user must enter an appropriate value. The following are the descriptions of available values.

- **yes**: Continue.
- **no**: Do not move forward with any stages from now on.
- **skip**: Skip to the next stage. This input value is used to ignore a stage that has not necessarily been executed when retrying the script after it has failed.

Constraints

- **Online backup of a master node**: Existing backups of master or the slave nodes cannot be used for replication rebuilding. You must use the online backup of the master node that is automatically created by the script.
- **Error while executing the replication rebuilding script**: The replication rebuilding script is not automatically rolled back to its previous stage, even when an error occurs during the execution. This is because the slave node cannot provide normal service before the replication rebuilding script is executed. To return to the stage in which the replication rebuilding script had not been executed, you must back up the **db_ha_apply_info** information, which is an internal catalog of the master and slave nodes, and the existing replication log, before executing the replication rebuilding script.

Operation Scenario during Read/Write Service

Because this operation scenario is not affected by service read/write, its impact on service during CUBRID operation is very limited. Operation scenarios during read/write service are divided into operation scenarios with failover and operation scenarios without failover.

Operation Scenarios without Failover

You can perform the following task without restarting a node after terminating the node in the CUBRID HA group.

Most common operation task	Scenario	Considerations
Online Backup	Operation task is performed at each master node and slave node each during operation.	Note that there may be a delay in the transaction of master node due to the operation task.
Schema change (excluding basic key change), index change, authorization change	When an operation task occurs at a master node, it is automatically replication reflected to a slave node.	Because replication log is copied and reflected to a slave node after an operation task is completed in a master node, operation task time is doubled. If operation task time becomes an issue, operation task may be performed in the following operation scenario where failover is used.
Add volume, Delete volume	Operation task is performed at each DB regardless of HA structure.	Note that there may be a delay in the transaction of master node due to the operation task. If operation task time becomes an issue, operation task may be performed in the following operation scenario where failover is used.
Failure node server replacement	It can be replaced without restarting the CUBRID HA group when a failure occurs.	The failure node must be registered in the ha_node_list of CUBRID HA group, and the

		node name must not be changed during replacement.
Failure broker server replacement	It can be replaced without restarting the broker when a failure occurs.	The connection to a broker replaced at a client can be made by rctime which is configured in URL string.
DB server expansion	You can execute cubrid heartbeat reload after configuration change (ha_node_list, ha_replica_list) without restarting the previously configured CUBRID HA group.	Note that all the management processes of a node are stopped when cubrid heartbeat reload is failed.
Broker server expansion	Run additional brokers without restarting existing brokers.	Modify the URL string to connect to a broker where a client is added.

Operation Scenario with Failover

Run the following task after completing the operation task following the stopping of a node in CUBRID HA group.

Most common operation task	Scenario	Considerations
DB server configuration change	A node whose configuration is changed is restarted when the configuration in cubrid.conf is changed.	
Change broker configuration, add broker, and delete broker	A broker whose configuration is changed is restarted when the configuration in cubrid_broker.conf is changed.	
DBMS version patch	Restart nodes and brokers in HA group after version patch.	Version patch means there is no change in the internal protocol, volume, and log of CUBRID.

Operation Scenario during Read Service

This operation scenario provides only the read service for operation tasks. It is necessary to allow only the read service, or to dynamically change the mode of the broker to Read Only. The operation scenario during the read service can be divided into two groups: operation scenarios with failover, and operation scenarios without failover.

Operation Scenario without Failover

The following task can be performed without terminating and restarting nodes in the CUBRID HA group.

Most common operation tasks	Scenario	Considerations
Schema change (primary key change)	When an operation task is performed at the master node, it is automatically reflected to the slave node.	In order to change the primary key, the existing key must be deleted and a new one added. For this reason, replication reflection may not occur due to the HA internal structure which reflects primary key-based replication logs. Therefore, operation tasks must be performed during the read service.

Operation Scenario with Failover

The following tasks must be started once all nodes in the CUBRID HA group are terminated and all operation tasks are complete.

Most common operation tasks	Scenario	Considerations

DBMS version upgrade	Restart each node and broker in the CUBRID HA group after they are upgraded.	A version upgrade means that there have been changes in the internal protocol, volume, or log of CUBRID. Because there are two different versions of the protocols, volumes, and logs of a broker and server during an upgrade, an operation task must be performed to make sure that each client and broker (before/after upgrade) are connected to the corresponding counterpart in the same version.
Schema change, index change, and authorization change	Stop the node that must be changed, perform an operation task, and then execute the node.	Although this operation task can be executed in an operation scenario without failover during the read and write service, it takes a long time to execute. Therefore, if operation task time becomes an issue, change each node while HA is stopped.
Massive data processing (INSERT / UPDATE / DELETE)	Stop the node that must be changed, perform an operation task, and then execute the node.	This processes massive data that cannot be segmented.

Operation Scenario after Service Stop

In this operation scenario, you must perform operation task after stopping all the nodes in the CUBRID HA group.

Most common operation tasks	Scenario	Considerations
Changing the host name and IP of a DB server	Stop all nodes in the CUBRID HA group, and restart them after the operation task.	When a host name has been changed, change the databases.txt of each broker and reset the broker connection with cubrid broker reset.

Performance Tuning

Performance Tuning

This chapter provides information about configuring system parameters that can affect the system performance. System parameters determine overall performance and operation of the system. This chapter explains how to use configuration files for database server and broker as well as a description of each parameter. For CUBRID Manager server configuration, see CUBRID Manager Manual.

This chapter covers the following topics:

- Configuring the Database server
- Configuring the Broker

Database Server Configuration

Scope of Database Server Configuration

CUBRID consists of the Database Server, the Broker and the CUBRID Manager. Each component has its configuration file. The system parameter configuration file for the Database Server is **cubrid.conf** located in the **$CUBRID/conf** directory. System parameters configured in **cubrid.conf** affect overall performance and operation of the database system. Therefore, it is very important to understand the Database Server configuration.

The CUBRID Database Server has a client/server architecture. To be more specific, it is divided into a Database Server process linked to the server library and a Broker process linked to the client library. The server process manages the database storage structure and provides concurrency and transaction functionalities. The client process prepares for query execution and manages object/schema.

System parameters for the database server, which can be set in the **cubrid.conf file**, are classified into a client parameter, a server parameter and a client/server parameter according to the range to which they are applied. A client parameter is only applied to client processes such as the broker. A server parameter affects the behaviors of the server processes. A client/server parameter must be applied to both the server and the client.

Location of cubrid.conf File and How It Works

- A Database Server process refers only to the **$CUBRID/conf/cubrid.conf** file. Database-specific configurations are distinguished by sections in the **cubrid.conf** file.
- A client process (i) refers to the **$CUBRID/conf/cubrid.conf** file and then (ii) additionally refers to the **cubrid.conf** file in the current directory (**$PWD**). The configuration of the file in the current directory (**$PWD/cubrid.conf**) overwrites that of the **$CUBRID/conf/cubrid.conf** file. That is, if the same parameter configuration exists in **$PWD/cubrid.conf** and in **$CUBRID/conf/cubrid.conf**, the configuration in **$PWD/cubrid.conf** has the priority.

cubrid.conf Configuration File and Default Parameters

CUBRID consists of the Database Server, the Broker and the CUBRID Manager. The name of the configuration file for each component is as follows. These files are all located in the **$CUBRID/conf** directory.

- Database Server configuration file : **cubrid.conf**
- Broker configuration file : **cubrid_broker.conf**
- CUBRID Manager server configuration file : **cm.conf**

cubrid.conf is a configuration file that sets system parameters for the CUBRID Database Server and determines overall performance and operation of the database system. In the **cubrid.conf** file, some important parameters needed for system installation are provided, having their default values.

Database Server System Parameters

The following are Database Server system parameters that can be used in the **cubrid.conf** configuration file. For the scope of **client** and **server parameters**, see Scope of Database Server Configuration.

You can change the parameters that are capable of dynamically changing the setting value through the **SET SYSTEM PARAMETERS** statement or a session command of CSQL interpreter, **;set** dynamically. If you are a DBA, you can change parameters regardless of the applied classification. However, if you are not a DBA, you can only change client parameters.

Purpose	Parameter Name	Scope	Type	Default Value	Dynamic Change
connection-related	cubrid_port_id	client parameter	int	1523	
	db_hosts	client parameter	string	NULL	O

	max_clients	server parameter	int	100	
memory-related	data_buffer_size	server parameter	int	512M	
	index_scan_oid_buffer_size	server parameter	int	64K	
	sort_buffer_size	server parameter	int	2M	
	temp_file_memory_size_in_pages	server parameter	int	4	
	thread_stack_size	server parameter	int	102400	
	garbage_collection	client parameter	bool	no	O
disk-related	temp_file_max_size_in_pages	server parameter	int	-1	
	temp_volume_path	server parameter	string	NULL	
	unfill_factor	server parameter	float	0.1	
	volume_extension_path	server parameter	string	NULL	
	dont_reuse_heap_file	server parameter	bool	no	
	db_volume_size	server parameter	int	512M	
	log_volume_size	server parameter	int	512M	
error message-related	call_stack_dump_activation_list	client/server parameter	string	NULL	
	call_stack_dump_deactivation_list	client/server parameter	string	NULL	
	call_stack_dump_on_error	client/server parameter	bool	no	O
	error_log	client/server parameter	string	cub_client.err, cub_server.err	
	error_log_level	client/server parameter	string	SYNTAX	O
	error_log_warning	client/server parameter	bool	no	O
	error_log_size	client/server parameter	int	8000000	O
concurrency/lock-related	deadlock_detection_interval_in_secs	server parameter	int	1	O
	isolation_level	client parameter	int	3	O
	lock_escalation	server parameter	int	100000	
	lock_timeout_in_secs	client parameter	int	-1	O
	lock_timeout_message_type	server parameter	int	0	O
logging-related	log_buffer_size	server parameter	int	2M	
	media_failure_support	server parameter	bool	yes	
	log_max_archives	server parameter	int	INT_MAX	O
	force_remove_log_archives	server parameter	bool	yes	O
	background_archiving	server parameter	bool	yes	O
	page_flush_interval_in_msecs	server parameter	int	0	O
	checkpoint_interval_in_mins	server parameter	int	720	O
	checkpoint_every_npages	server parameter	int	10000	
	adaptive_flush_control	server parameter	bool	yes	O

	max_flush_pages_per_second	server parameter	int	10000	O
	sync_on_nflush	server parameter	int	200	O
transaction processing-related	async_commit	server parameter	bool	no	
	group_commit_interval_in_msecs	server parameter	int	0	O
statement/type-related	ansi_quotes	client parameter	bool	yes	
	block_ddl_statement	client parameter	bool	no	O
	block_nowhere_statement	client parameter	bool	no	O
	compat_numeric_division_scale	client/server parameter	bool	no	O
	intl_mbs_support	client parameter	bool	no	
	oracle_style_empty_string	client parameter	bool	no	
	only_full_group_by	client parameter	bool	no	O
	pipes_as_concat	client parameter	bool	yes	
	add_column_update_hard_default	client parameter	bool	no	O
	plus_as_concat	client parameter	bool	yes	
	return_null_on_function_errors	client/server parameter	bool	no	O
	no_backslash_escapes	client parameter	bool	yes	
	require_like_escape_character	client parameter	bool	no	
	alter_table_change_type_strict	client/server parameter	bool	no	O
	default_week_format	client/server parameter	int	0	O
	group_concat_max_len	server parameter	int	1024	O
query cache-related	max_plan_cache_entries	client/server parameter	int	1000	
	max_query_cache_entries	server parameter	int	-1	
	query_cache_mode	server parameter	int	0	
	query_cache_size_in_pages	server parameter	int	-1	
utility-related	compactdb_page_reclaim_only	server parameter	int	0	
	csql_history_num	client parameter	int	50	O
	communication_histogram	client parameter	bool	no	O
	backup_volume_max_size_bytes	server parameter	int	-1	
HA-related	ha_mode	server parameter	string	off	
other	service	server parameter	string		
	server	server parameter	string		
	index_scan_in_oid_order	client parameter	bool	no	O
	single_byte_compare	server parameter	bool	no	
	insert_execution_mode	client parameter	int	1	
	java_stored_procedure	server parameter	bool	no	
	pthread_scope_process	server parameter	bool	yes	

	auto_restart_server	server parameter	bool	yes	O
	index_unfill_factor	server parameter	float	0.05	
	use_orderby_sort_limit	server parameter	bool	yes	O
	session_state_timeout	server parameter	int	21600	
	multi_range_optimization_limit	server parameter	int	100	O
	access_ip_control	server parameter	bool	no	
	access_ip_control_file	server parameter	string		

Section by Parameter

Parameters specified in **cubrid.conf** have the following three sections:

- Used when the CUBRID service starts : [service] section
- Applied commonly to all databases : [common] section
- Applied individually to each database : [@<*database*>] section

Where <*database*> is the name of the database to which each parameter applies. If a parameter configured in [common] is the same as the one configured in [@<*database*>], the one configured in [@<*database*>] is applied.

Default Parameters

cubrid.conf, a default database configuration file created during the CUBRID installation, includes some default Database Server parameters that must be changed. You can change the value of a parameter that is not included as a default parameter by manually adding or editing one.

The following is the content of the **cubrid.conf** file.

```
# Copyright (C) 2008 Search Solution Corporation. All rights reserved by Search Solution.
#
# $Id$
#
# cubrid.conf#
# For complete information on parameters, see the CUBRID
# Database Administration Guide chapter on System Parameters
# Service section - a section for 'cubrid service' command
[service]
# The list of processes to be started automatically by 'cubrid service start' command
# Any combinations are available with server, broker and manager.
service=server,broker,manager
# The list of database servers in all by 'cubrid service start' command.
# This property is effective only when the above 'service' property contains 'server'
keyword.
#server=foo,bar
# Common section - properties for all databases
# This section will be applied before other database specific sections.
[common]
# Number of data buffer pages
# data_buffer_pages (25,000 pages) * DB page size (4KB) = 100M
data_buffer_pages=25000
# Number of sort buffer pages
# sort_buffer_pages (16 pages) * DB page size (4KB) * number of threads
sort_buffer_pages=16
# Number of log buffer pages.
# log_buffer_pages (50 pages) * DB page size (4KB) = 200KB
log_buffer_pages=50
# Maximum number of locks acquired on individual instances of a
# class before the locks on the instances are escalated to a class lock
lock_escalation=100000
# Minimal amount of time to wait for a lock (seconds).
# A negative value, indicates to wait indefinitely until the lock is
# granted or until the transaction is rolled back as a result of a deadlock.
# A value of 0, indicates not to wait for a lock.
lock_timeout_in_secs=-1
# Interval between attempts at deadlock detection (seconds).
# An approximate interval to attempt to run the deadlock detector.
```

```
deadlock_detection_interval_in_secs=1
# Checkpoint when the specified time has passed (minutes).
# Checkpoint will be done also when log has grown by specified pages.
checkpoint_interval_in_mins=720
# Transaction isolation level.
# Six levels of isolation are provided, represented by:
# "TRAN_SERIALIZABLE"
# "TRAN_REP_CLASS_REP_INSTANCE"
# "TRAN_REP_CLASS_COMMIT_INSTANCE"
# "TRAN_REP_CLASS_UNCOMMIT_INSTANCE"
# "TRAN_COMMIT_CLASS_COMMIT_INSTANCE"
# "TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE"
# For other aliases, or for more information on the levels, see the
# System Parameters chapter in the Database Administration Guide.
isolation_level="TRAN_REP_CLASS_UNCOMMIT_INSTANCE"
# TCP port id for the CUBRID programs (used by all clients).
cubrid_port_id=1523
# The maximum number of concurrent client connections the server will accept.
# This value also means the total # of concurrent transactions.
max_clients=50
# Restart the server process automatically
auto_restart_server=yes
# Become a master server for replication.
replication=no
# Enable Java Stored Procedure
java_stored_procedure=no
```

Connection-Related Parameters

The following are parameters related to the Database Server. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
cubrid_port_id	int	1523	1	
db_hosts	string	NULL		
max_clients	int	100	10	1024

cubrid_port_id

cubrid_port_id is a parameter that configures the port to be used by the master process. The default value is **1,523**. If the port 1,523 is already being used on the server where CUBRID is installed or it is blocked by a firewall, an error message, which means the master server is not connected because the master process cannot be running properly, is outputted. If such port conflict occurs, the administrator must change the value of **cubrid_port_id** considering the server environment.

db_hosts

db_hosts is a parameter that specifies a list of Database Server hosts to which clients can connect, and the connection order. The server host list consists of more than one server host names, and host names are separated by spaces or colons (:). Duplicate or non-existent names are ignored.

The following is an example that shows the values of the **db_hosts** parameter. In this example, connections are attempted in the order of **host1** > **host2** > **host3**.

```
db_hosts="hosts1:hosts2:hosts3"
```

To connect to the server, the client first tries to connect to the specified server host referring to the database location file (**databases.txt**). If the connection fails, the client then tries to connect to the first one of the secondarily specified server hosts by referring to the value of the **db_hosts** parameter in the database configuration file (**cubrid.conf**).

max_clients

max_clients is a parameter that configures the maximum number of clients (usually Broker processes) which allow concurrent connections to the database server. The **max_clients** parameter refers to the number of concurrent transactions. The default value is **100**.

To grantee performance while increasing the number of concurrent users in CUBRID environment, you need to assign an approviate value to **max_clients** (**cubrid.conf**) and **MAX_NUM_APPL_SERVER** (**cubrid_broker.conf**) parameters. That is, you are required to modify the number of concurrent connections allowed by databases with the **max_clients** parameter. You should also modify the number of concurrent connections allowed by brokers with the **MAX_NUM_APPL_SERVER** parameter.

For example, in the **cubrid_broker.conf** file, the **MAX_NUM_APPL_SERVER** value of [%query_editor] is 50 and the **MAX_NUM_APPL_SERVER** value of [%BROKER1] is 50, you should specify the **max_clients** parameter value as 120 (100 multiplied by 2) in the **cubrid.conf** file so that it can have more free space.

Memory-Related Parameters

The following are parameters related to the memory used by the Database Server or client. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
data_buffer_size	int	512M	64K	
index_scan_oid_buffer_size	int	64K	1K	256K
sort_buffer_size	int	2M	64K	
temp_file_memory_size_in_pages	int	4	0	20
thread_stacksize	int	102400	65536	
garbage_collection	bool	no		

data_buffer_size

data_buffer_size is a parameter that configures the number of data pages to be cached in the memory by the Database Server. You can set units as K, M, G and T, which stand for KB (kilobytes), MB (megabytes), GB (gigabytes) and TB (terabytes), respectively. If you omit the unit, bytes will be applied. The default value is 512M, and the minimum value is 64K.

The greater the value of the **data_buffer_size** parameter, the more data pages to be cached in the buffer, thus providing the advantage of decreased disk I/O cost. However, if this parameter is too large, the buffer pool can be swapped out by the operating system because the system memory is excessively occupied. It is recommended to configure the **data_buffer_size** parameter in a way the required memory size is less than two-thirds of the system memory size.

- Required memory size = the number of buffer size (**data_buffer_size**)

index_scan_oid_buffer_size

index_scan_oid_buffer_size is a parameter that configures the number of buffer pages where the OID list is to be temporarily saved during the index scan. You can set units as K, M, G and T, which stand for KB (kilobytes), MB (megabytes), GB (gigabytes) and TB (terabytes), respectively. If you omit the unit, bytes will be applied. The default value is 2M, and the minimum value is 64K.

The size of the OID buffer tends to vary in proportion to the value of the **index_scan_oid_buffer_size** parameter and the page size set when the database was created. In addition, the bigger the size of such OID buffer, the more the index scan cost. You can set the value of the **index_scan_oid_buffer_size** by considering these factors.

sort_buffer_size

sort_buffer_size is a parameter that configures the number of buffer pages to be used when sorting. You can set units as K, M, G and T, which stand for KB (kilobytes), MB (megabytes), GB (gigabytes) and TB (terabytes), respectively. If you omit the unit, bytes will be applied. The default value is 2M, and the minimum value is 64K.

The server assigns one sort buffer for each client request, and releases the assigned buffer memory when sorting is complete.

temp_file_memory_size_in_pages

temp_file_memory_size_in_pages is a parameter that configures the number of buffer pages to cache temporary result of a query. The default value is **4** and the maximum value is 20.

- Required memory size = the number of temporary memory buffer pages (**temp_file_memory_size_in_pages** * **page size**)
- The number of temporary memory buffer pages = the value of the **temp_file_memory_size_in_pages** parameter
- Page size = the value of the page size specified by the **-s** option of the **cubrid createdb** utility during the database creation

thread_stacksize

thread_stacksize is a parameter that configures the stack size of a thread. The default value is **100*1024**. The value of the **thread_stacksize** parameter must not exceed the stack size allowed by the operating system.

garbage_collection

garbage_collection is a parameter that specifies whether or not to collect garbage memory no longer used by the client. The default value is **no**.

Disk-Related Parameters

The following are disk-related parameters for defining database volumes and saving files. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
temp_file_max_size_in_pages	int	-1		
temp_volume_path	string	NULL		
unfill_factor	float	0.1	0.0	0.3
dont_reuse_heap_file	bool	no		
db_volume_size	int	512M	20M	20G
log_volume_size	int	512M	20M	4G

temp_file_max_size_in_pages

temp_file_max_size_in_pages is a parameter that configures the maximum number of pages to store temporary volumes in the disk, which are used for the execution of complex queries or sorting. The default value is **-1**. If this parameter is configured to the default value, unlimited number of temporary volumes are created and stored in the directory specified by the **temp_volume_path** parameter. If it is configured to 0, the administrator must create temporary volumes manually by using the **cubrid addvoldb** utility because temporary volumes are not created automatically.

temp_volume_path

temp_volume_path is a parameter that specifies the directory in which to create temporary volumes used for the execution of complex queries or sorting. The default value is the volume location configured during the database creation.

volume_extension_path

volume_extension_path is a parameter that specifies the directory in which to create extra volumes if **cubrid addvoldb** is used without **-F** option that specifies the extra volume path. The default value is the volume location configured during the database creation.

unfill_factor

unfill_factor is a parameter that defines the rate of disk space to be allocated in a heap page for data updates. The default value is **0.1**. That is, the rate of free space is configured to 10%. In principle, data in the table is inserted in physical order. However, if the size of the data increases due to updates and there is not enough space for storage in the given page, performance may degrade because updated data must be relocated to another page. To prevent such a problem, you can configure the rate of space for a heap page by using the **unfill_factor** parameter. The allowable maximum value is 0.3 (30%). In a database where data updates rarely occur, you can configure this parameter to 0.0 so that space will not be allocated in a heap page for data updates. If the value of the **unfill_factor** parameter is negative or greater than the maximum value, the default value (**0.1**) is used.

volume_extension_path

volume_extension_path is a parameter that specifies the directory where automatically extended volumes are to be created. The default value is the volume location configured during the database creation.

dont_reuse_heap_file

The parameter "**dont_reuse_heap_file**" specifies whether or not heap files, which are deleted when deleting the table (DROP TABLE), are to be reused when creating a new table (CREATE TABLE). If this parameter is set to 0, the deleted heap files can be reused; if it is set to 1, the deleted heap files are not used when creating a new table. The default value is 0.

db_volume_size

db_volume_size is a parameter that defines the following values. The default value is **512M**.

- The default database volume size when **cubrid createdb** and **cubrid addvoldb** utility is used without **--db-volume-size** option.
- The default size of generic volume that is added automatically when database volume is full.

log_volume_size

log_volume_size is a parameter that defines the default size of log volume file when **cubrid createdb** utility is used without **--log-volume-size** option. You can set units as K, M, G and T, which stand for KB(kilobytes), MB(megabytes), GB(gigabytes) and TB(terabytes), respectively. If you omit the unit, bytes will be applied. The default value is **512M**.

Error Message-Related Parameters

The following are parameters related to processing error messages recorded by CUBRID. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value
call_stack_dump_activation_list	string	NULL
call_stack_dump_deactivation_list	string	NULL

call_stack_dump_on_error	bool	no
error_log	string	cub_client.err, cub_server.err
error_log_level	string	SYNTAX
error_log_warning	bool	no
error_log_size	int	8000000

call_stack_dump_activation_list

call_stack_dump_activation_list is a parameter that specifies a certain error number for which a call stack is to be dumped as an exception even when you configure that a call stack will not be dumped for any errors. Therefore, the **call_stack_dump_activation_list** parameter is effective only when **call_stack_dump_on_error=no**. The following is an example that configures the parameter so that call stacks will not be dumped for any errors, except for the ones whose numbers are -115 and -116.

```
call_stack_dump_on_error= no
call_stack_dump_activation_list=-115,-116
```

call_stack_dump_deactivation_list

call_stack_dump_deactivation_list is a parameter that specifies a certain error number for which a call stack is not to be dumped when you configure that a call stack will be dumped for any errors. Therefore, the **call_stack_dump_deactivation_list** parameter is effective only when **call_stack_dump_on_error=yes**. The following is an example that configures the parameter so that call stacks will be dumped for any errors, except for the ones whose numbers are -115 and -116.

```
call_stack_dump_on_error= yes
call_stack_dump_deactivation_list=-115,-116
```

call_stack_dump_on_error

call_stack_dump_on_error is a parameter that determines whether or not to dump a call stack when an error occurs in the Database Server. If this parameter is configured to no, a call stack for any errors is not dumped. If it is configured to yes, a call stack for all errors is dumped. The default value is **no**.

error_log

error_log is a server/client parameter that specifies the name of the error log file when an error occurs in the database server. The name of the error log file must be in the form of <*database_name*>_<*date*>_<*time*>**.err**. However, the naming rule of the error log file does not apply to errors for which the system cannot find the Database Server information. Therefore, error logs are recorded in the **cubrid.err** file. The error log file **cubrid.err** is stored in the **$CUBRID/log/server** directory.

error_log_level

error_log_level is a server parameter that specifies a error message to be stored based on severity. There are five different levels which ranges from **NOTIFICATION** (lowest level), **WARNING**, **SYNTAX**, **ERROR**, and **SYNTAX** (highest level). An error message with **SYNTAX**, **ERROR**, and FATAL levels are stored in the log file if severity of error is **SYNTAX**, default value.

error_log_warning

The server parameter **error_log_warning** specifies whether or not error messages with a severity level of **WARNING** are to be displayed. Its default value is no. Therefore, only error messages with levels other than **WARNING** will be saved even when it is set to **error_log_level = NOTIFICATION**. For this reason, you must set **error_log_warning = yes** to save WARNING messages to an error log file.

error_log_size

error_log_size is a parameter that specifies the maximum number of lines per an error log file. The default value is **8,000,000**. If it reaches up the specified number, the <*database_name*>_<*date*>_<*time*>.err.bak file is created.

Concurrency/Lock Parameters

The following are parameters related to concurrency control and locks of the Database Server. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
deadlock_detection_interval_in_secs	int	1	1	
isolation_level	int	3	1	6
lock_escalation	int	100000	5	
lock_timeout_in_secs	int	-1	-1	
lock_timeout_message_type	int	0	0	2

deadlock_detection_interval_in_secs

deadlock_detection_interval_in_secs is a parameter that configures the interval (in seconds) in which deadlocks are detected for stopped transactions. If a deadlock occurs, CUBRID resolves the problem by rolling back one of the transactions. The default value is 1 second. Note that deadlocks cannot be detected if the detection interval is too long.

isolation_level

isolation_level is a parameter that configures the isolation level of a transaction. The higher the isolation level, the less concurrency and the less interruption by other concurrent transactions. The **isolation_level** parameter can be configured to an integer value from 1 to 6, which represent isolation levels, or character strings. The default value is **TRAN_REP_CLASS_UNCOMMIT_INSTANCE**. For details about each isolation level and parameter values, see Setting Isolation Level and the following table.

Isolation Level	isolation_level Parameter Value
SERIALIZABLE	"TRAN_SERIALIZABLE" or 6
REPEATABLE READ CLASS with REPEATABLE READ INSTANCES	"TRAN_REP_CLASS_REP_INSTANCE" or "TRAN_REP_READ" or 5
REPEATABLE READ CLASS with READ COMMITTED INSTANCES(or CURSOR STABILITY)	"TRAN_REP_CLASS_COMMIT_INSTANCE" or "TRAN_READ_COMMITTED" or "TRAN_CURSOR_STABILITY" or 4
REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES	"TRAN_REP_CLASS_UNCOMMIT_INSTANCE" or "TRAN_READ_UNCOMMITTED" or 3
READ COMMITTED CLASS with READ COMMITTED INSTANCES	"TRAN_COMMIT_CLASS_COMMIT_INSTANCE" or 2
READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES	"TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE" or 1

- **TRAN_SERIALIZABLE** : This isolation level ensures the highest level of consistency. For more information, see SERIALIZABLE.
- **TRAN_REP_CLASS_REP_INSTANCE** : This isolation level can occur phantom read. For more information, see REPEATABLE READ CLASS with REPEATABLE READ INSTANCES.

- **TRAN_REP_CLASS_COMMIT_INSTANCE** : This isolation level can occur unrepeatable read. For more information, see REPEATABLE READ CLASS with READ COMMITTED INSTANCES.
- **TRAN_REP_CLASS_UNCOMMIT_INSTANCE** : This isolation level can occur dirty read. For more information, see REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES.
- **TRAN_COMMIT_CLASS_COMMIT_INSTANCE** : This isolation level can occur unrepeatable read. It allows modification of table schema by current transactions while data is being retrieved. For more information, see READ COMMITTED CLASS with READ COMMITTED INSTANCES.
- **TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE** : This isolation level can occur dirty read. It allows modification of table schema by current transactions while data is being retrieved. For more information, see READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES.

lock_escalation

lock_escalation is a parameter that specifies the maximum number of locks permitted before row level locking is extended to table level locking. The default value is **100,000**. If the value of the **lock_escalation** parameter is small, the overhead by memory lock management is small as well; however, the concurrency decreases. On the other hand, if the configured value is large, the overhead is large as well; however, the concurrency increases.

lock_timeout_in_secs

lock_timeout_in_secs is a client parameter that configures the lock waiting time. If the lock is not permitted within the specified time period, the given transaction is canceled, and an error message is returned. If the parameter is configured to **-1**, which is the default value, the waiting time is infinite until the lock is permitted. If it is configured to 0, there is no waiting for locks.

lock_timeout_message_type

lock_timeout_message_type is a parameter that configures the level of information that is to be included in the message returned when a lock timeout occurs. If the parameter is configured to **0**, which is the default value, the information about lock ownership is not included in the message. If it is configured to 1, single lock ownership information is included. If it is configured to 2, all information about lock ownership is included.

- If **lock_timeout_message_type** = 0

```
ERROR: Your transaction (index 3, cub_user@cdbs006.cub|15668) timed out waiting
on    X_LOCK lock on instance 0|636|34 of class participant. You are waiting for
user(s)   to finish.
```

- If **lock_timeout_message_type** = 1

```
ERROR : Your transaction (index 3, cub_user@cdbs006.cub|15668) timed out waiting
on    X_LOCK lock on instance 0|636|34 of class participant. You are waiting for user(s)
cub_user@cdbs006.cub|15615 to finish.
```

- If **lock_timeout_message_type** = 2

```
ERROR: Your transaction (index 3, cub_user@cdbs006.cub|15668) timed out waiting
on    X_LOCK lock on instance 0|636|34 of class participant. You are waiting for user(s)
cub_user@cdbs006.cub|15615, cub_user@cdbs006.cub|15596 to finish.
```

Logging-Related Parameters

The following are parameters related to logs used for database backup and restore. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
log_buffer_size	int	2M	192K	
media_failure_support	bool	yes		
log_max_archives	int	INT_MAX	0	
force_remove_log_archives	bool	yes		
background_archiving	bool	yes		

page_flush_interval_in_msecs	int	0	-1	
checkpoint_interval_in_mins	int	720	1	
checkpoint_every_npages	int	10000	10	
adaptive_flush_control	bool	yes		
max_flush_pages_per_second	int	10000	1	INT_MAX
sync_on_nflush	int	200	1	INT_MAX

log_buffer_size

log_buffer_size is a parameter that configures the number of log buffer pages to be cached in the memory. The default value is 2M. If the value of the **log_buffer_size** parameter is big, performance can be improved (due to the decrease in disk I/O) when transactions are long and numerous. It is recommended to configure an appropriate value considering the memory size and operations of the system where CUBRID is installed.

- Required memory size = the number of log buffer size (**log_buffer_size**)

media_failure_support

media_failure_support is a parameter that specifies whether or not to store archive logs in case of storage media failure. If the parameter is configured to **yes**, which is the default value, all active logs are copied to archive logs when the active logs are full and the transaction is active. If it is configured to no, archive logs created after the active logs are full are deleted automatically. Note that archive logs are deleted automatically if the value of the parameter is configured to no.

If you specify this parameter to no, the **backgroud_archiving** parameter is deactivated, accordingly.

log_max_archives

log_max_archives is a parameter that sets the maximum number of archive log files to record if **media_failure_support** is set to yes. The minimum value is set to zero, and the default is **INT_MAX**. The operation of this parameter differs depending on the configuration of **force_remove_log_archives**.

For example, when **log_max_archives** is 3 and **force_remove_log_archives** is yes in cubrid.conf, the most recent three archive log files are recorded. If a fourth archiving log file is generated, the oldest archive log file is automatically deleted. The information about the deleted archive log is recorded into the *_login file.

However, if an active transaction still refers to an existing archive log file, the archive log file will not be deleted. That is, if a transaction starts at the point that the first archive log file is generated, and it is still active until the fifth archive log is generated, the first archive log file cannot be deleted.

For how to set up the CUBRID HA environment, see cubrid.conf.

force_remove_log_archives

The **force_remove_log_archives** parameter is used to determine whether to allow the deletion of the files other than the recent log archive files of which the number is specified by **log_max_archives**, and the default value is yes.

If you set yes for the parameter, the files will be deleted other than the recent log archive files for which the number is specified by **log_max_archives**.

If you set no for the parameter, the log archive files will not be deleted. However, if you set for **ha_mode** to on, the files other than the log archive files required for the HA related process and the recent log archive files of which the number is specified by **log_max_archives** will be deleted.

If you want to build a CUBRID HA environment, see Configuration.

background_archiving

background_archiving is a parameter that generate a temporary archive log periodically at a specific time if **media_failure_support** is set to yes. This is useful when balancing disk I/O load due to the archive log process. The default is **yes**.

checkpoint_interval_in_mins

checkpoint_interval_in_mins is a parameter that sets cycle (in minutes) for checkpoint to be executed. The default value is **720.**

Checkpoint flushes log files(dirty page) remained in data buffers to a disk. It can restore data back to the latest checkpoint if failure happens. If high volume of log files are stored in a disk due to checkpoint, it may cause disk I/O. Therefore, you should set the checkpoint cycle properly to prevent database operation failure.

The **checkpoint_interval_in_mins** and **checkpoint_every_npages** parameters are related to setting checkpoint cycle. The checkpoint is periodically executed whenever the time specified in **checkpoint_interval_in_mins** parameter has elapsed or the number of log pages specified in **checkpoint_every_npages** parameter has reached.

checkpoint_every_npages

checkpoint_every_npages is a parameter that sets checkpoint cycle by log page. The default value is **10,000**. You can distribute disk I/O overload at the checkpoint by specifying lower number in the **checkpoint_every_npages** parameter, especially in the environment where **INSERT**/**UPDATE** are heavily loaded at a specific time.

page_flush_interval_in_msecs

The parameter **page_flush_interval_in_msecs** specifies the interval in milliseconds (msec) at which dirty pages in a data buffer are flushed to a disk. Its default value is 0. If this parameter is set to -1 (the minimum value), dirty pages are flushed to the disk only at the checkpoint, or when pages are swapped.

This is a parameter that is related to I/O load and buffer concurrency. For this reason, you must set its value in consideration of the workload of the service environment.

adaptive_flush_control

The parameter **adaptive_flush_control** automatically adjusts the flush capacity at every 50 ms depending on the current status of the flushing operation. Its default value is **yes**. That is, this capacity is increased if a large number of **INSERT** or **UPDATE** operations are concentrated at a certain point of time and the number of flushed pages reaches the **max_flush_pages_per_second** parameter value; and is decreased otherwise. In the same way, you can distribute the I/O load by adjusting the flush capacity on a regular basis depending on the workload.

max_flush_pages_per_second

The parameter **max_flush_pages_per_second** specifies the maximum flush capacity when the flushing operation is performed from a buffer to a disk. Its default value is 10,000. That is, you can prevent concentration of I/O load at a certain point of time by setting this parameter to control the maximum flush capacity per second.

If a large number of **INSERT** or **UPDATE** operations are concentrated at a certain point of time, and the flush capacity reaches the maximum capacity set by this parameter, only log pages are flushed to the disk, and data pages are no longer flushed. Therefore, you must set an appropriate value for this parameter considering the workload of the service environment.

sync_on_nflush

The parameter **sync_on_nflush** sets the interval in pages between after data and log pages are flushed from buffer and before they are synchronized with FILE I/O of operating system. Its default value is 200. That is, the CUBRID Server performs synchronization with the FILE I/O of the operating system whenever 200 pages have been flushed. This is also a parameter related to I/O load.

Transaction Processing-Related Parameters

The following are parameters for improving transaction commit performance. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
async_commit	bool	no		
group_commit_interval_in_msecs	int	0	0	

async_commit

async_commit is a parameter that activates the asynchronous commit functionality. If the parameter is configured to **no**, which is the default value, the asynchronous commit is not performed; if it is configured to yes, the asynchronous commit is executed. The asynchronous commit is a functionality that improves commit performance by completing the commit for the client before commit logs are flushed on the disk and having the log flush thread (LFT) perform log flushing in the background. Note that already committed transactions cannot be restored if a failure occurs on the Database Server before log flushing is performed.

group_commit_interval_in_msecs

group_commit_interval_in_msecs is a parameter that configures the interval (in milliseconds), at which the group commit is to be performed. If the parameter is configured to **0**, which is the default value, the group commit is not performed. The group commit is a functionality that improves commit performance by combining multiple commits that occurred in the specified time period into a group so that commit logs are flushed on the disk at the same time.

Statement/Type-Related Parameters

The following are parameters related to SQL statements and data types supported by CUBRID. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value
ansi_quotes	bool	yes
block_ddl_statement	bool	no
block_nowhere_statement	bool	no
compat_numeric_division_scale	bool	no
intl_mbs_support	bool	no
oracle_style_empty_string	bool	no
only_full_group_by	bool	no
pipes_as_concat	bool	yes
alter_table_change_type_strict	bool	no
default_week_format	bool	yes
group_concat_max_len	bool	no

ansi_quotes

ansi_quotes is a parameter that enclose symbols and character string to handle identifiers. The default value is **yes**. If this parameter value is set to **yes**, double quotations are handled as identifier symbols and single quotations are handled as character string symbols. If it is set to **no**, double quotations are handled as character string symbols.

block_ddl_statement

block_ddl_statement is a parameter that restricts the execution of DDL (Data Definition Language) statements by the client. If the parameter is configured to no, the given client is allowed to execute DDL statements. If it is configured to yes, the client is not permitted to execute DDL statements. The default value is **no**.

block_nowhere_statement

block_nowhere_statement is a parameter that restricts the execution of **UPDATE**/**DELETE** statements without a condition clause (**WHERE**) by the client. If the parameter is configured to no, the given client is allowed to execute **UPDATE**/**DELETE** statements without a condition clause. If it is configured to yes, the client is not permitted to execute **UPDATE**/**DELETE** statements without a condition clause. The default value is **no**.

compact_numeric_division_scale

compat_numeric_division_scale is a parameter that configures the scale to be displayed in the result (quotient) of a division operation. If the parameter is configured to **no**, the scale of the quotient is 9 if it is configured to **yes**, the scale is determined by that of the operand. The default value is **no**.

intl_mbs_support

intl_mbs_support is a parameter that specifies whether or not to support multi-byte character set. If the parameter is configured to **no**, a multi-byte character set is not allowed if it is configured to yes, a multi-byte character set is allowed. To improve performance, it is recommended to configure the **intl_mbs_support** parameter to **no** and use alphabets for table and column names because operation cost for supporting multi-byte character set is high.

oracle_style_empty_string

oracle_style_empty_string is a parameter that improves compatibility with other DBMS (Database Management Systems) and specifies whether or not to process empty strings as **NULL** as in Oracle DBMS. If the **oracle_style_empty_string** parameter is configured to **no**, the character string is processed as a valid string if it is configured to **yes**, the empty string is processed as **NULL**.

only_full_group_by

only_full_group_by is a parameter that specifies whether extended syntax about using **GROUP BY** statement is used or not.

If this parameter value is set to **no**, an extended syntax is applied thus, a column that is not specified in the **GROUP BY** statement can be specified in the **SELECT** column list. If it is set to **yes**, a column that is only specified in the **GROUP BY** statement can be the **SELECT** column list.

The default value is no. Therefore, specify the **only_full_group_by** parameter value to **yes** to execute queries by SQL standards. Because the extended syntax is not applied in this case, an error below is displayed.

```
ERROR: Attributes exposed in aggregate queries must also appear in the group by clause.
```

pipes_as_concat

pipes_as_concat is a parameter about using a double pipe symbol. The default value is **yes**. If this parameter value is set to **yes**, a double pipe symbol is handled as a concatenation operator if no, it is handled as the **OR** operator.

add_column_update_hard_default

The **add_column_update_hard_default** parameter is used to determine whether or not to provide the hard_default value as the input value for a column when you add a new column to the **ALTER TABLE … ADD COLUMN** clause.

If a value for this parameter is set to yes, enter a new input value of a column as a hard default value when you have **NOT NULL** constraints but no **DEFAULT** constraints. If the parameter value is set to no, enter **NULL**, even if **NOT NULL** constraints exist. If a value for this parameter is set to yes and there is no hard default value for the column type to add, an error message will be displayed and a roll-back occurs. For the hard default for each type, see the CHANGE Clause of the **ALTER TABLE** statement.

```
-- add_column_update_hard_default=no

CREATE TABLE tbl (i INT);
INSERT INTO tbl VALUES (1),(2);
ALTER TABLE tbl ADD COLUMN j INT NOT NULL;

SELECT * FROM TBL;

            i          j
========================
            2       NULL
            1       NULL

-- add_column_update_hard_default=yes

CREATE TABLE tbl (i int);
INSERT INTO tbl VALUES (1),(2);
ALTER TABLE tbl ADD COLUMN j INT NOT NULL;

SELECT * FROM tbl;

            i          j
=========================
            2          0
            1          0
```

plus_as_concat

The **plus_as_concat** parameter is a parameter for the use of the + operator, and the default value is yes. If a value for this parameter is set to yes, the + operator will be interpreted as a concatenation operator; if it is set to no, the operator will be interpreted as a numeric operator.

```
-- plus_as_concat = yes
SELECT '1'+'1';
         '1'+'1'
======================
         '11'  SELECT '1'+'a';

         '1'+'a'
======================
         '1a'

-- plus_as_concat = no
SELECT '1'+'1';
                '1'+'1'
==========================
 2.000000000000000e+000

SELECT '1'+'a';

ERROR: Cannot coerce 'a' to type double.
```

return_null_on_function_errors

The **return_null_on_function_errors** parameter is used to define actions when errors occur in some SQL functions, and the default value is no. If a value for this parameter is set to yes, **NULL** is returned; if it is set to no, an error is returned when the error occurs in functions, and the related message is displayed.

The following SQL functions are affected by this system parameter.

- YEAR
- MONTH
- DAY
- DAYOFMONTH
- HOUR

* MINUTE
* SECOND
* QUARTER
* WEEK
* WEEKDAY
* DAYOFWEEK
* DAYOFYEAR
* SEC_TO_TIME
* TIME_TO_SEC
* TO_DAYS
* FROM_DAYS
* MAKEDATE
* MAKETIME
* TIME
* FROM_UNIXTIME
* TIMEDIFF

```
-- return_null_on_function_errors=no

SELECT HOUR('2010-01-01');

ERROR: Conversion error in time format.

-- return_null_on_function_errors=yes

SELECT HOUR('2010-01-01');

   hour('2010-01-01')
======================
  NULL
```

no_backslash_escapes

The **no_backslash_escapes** is used to determine whether or not to use backslash (\) as an escape character, and the default value is yes. If a value for this parameter is set to no, backslash (\) will be used as an escape character; if it is set to yes, backslash (\) will be used as a normal character. For more information, see Special Character Escape.

require_like_escape_character

The **require_like_escape_character** parameter is used to determine whether or not to use an ESCAPE character in the **LIKE** clause, and the default value is no. If a value for this parameter is set to yes and a value for **no_backslash_escapes** is set to no, backslash (\) will be used as an ESCAPE character in the strings of the LIKE clause, otherwise you should specify an ESCAPE character by using the **LIKE… ESCAPE** clause. For more information, see LIKE Predicate.

alter_table_change_type_strict

The **alter_table_change_type_strict** parameter is used to determine whether or not to allow the conversion of column values according to the type change, and the default value is no. If a value for this parameter is set to no, the value may be changed when you change the column types or when you add **NOT NULL** constraints; if it is set to yes, the value is not changed. For more information, see CHANGE Clause in the **ALTER TABLE** statement.

default_week_format

The **default_week_format** parameter is used to set the default value for the *mode* attribute of the WEEK function. The default value is 0. For more information, see WEEK Function.

group_concat_max_len

The **group_concat_max_len** parameter is used to limit the return value size of the **GROUP_CONCAT** function. The default value is 1024 bytes, the minimum value is 4 bytes, and the maximum value is 33,554,432 bytes. If the return value of the **GROUP_CONCAT** function exceeds the limitation, **NULL** will be returned.

Query Cache-Related Parameters

The following are parameters related to the query cache functionality that provides execution results cached for the same **SELECT** statement. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
max_plan_cache_entries	int	1,000		
max_query_cache_entries	int	-1		
query_cache_mode	int	0	0	2
query_cache_size_in_pages	int	-1		

max_plan_cache_entries

max_plan_cache_entries is a parameter that configures the maximum number of query plans to be cached in the memory. If the **max_plan_cache_entries** parameter is configured to -1 or 0, generated query plans are not stored in the memory cache; if it is configured to an integer value equal to or greater than 1, a specified number of query plans are cached in the memory. Also, the value of this parameter must be configured to an integer value equal to or greater than 1 to use the query cache functionality that caches the results of the same query.

max_query_cache_entries

max_query_cache_entries is a parameter that configures the maximum number of query results to be cached. If the parameter is configured to -1 or 0, the query cache functionality is deactivated; if it is configured to an integer value equal to or greater than 1, the execution results of a specified number of queries are cached. With the query cache functionality, you can expect performance improvement in cases where query data does not change, and the same query is entered repeatedly. Note that the query cache functionality is activated only when the **max_plan_cache_entries** parameter, which activates the query plan cache functionality, is configured to an integer value equal to or greater than 1 because the query cache functionality is dependent of the query plan cache functionality.

query_cache_mode

query_cache_mode is a parameter that specifies one of two query cache modes. In the primary query cache mode, all queries are cached. In the second query cache mode, the query with the hint /*+QUERY_CACHE(1) */ is only cached. If this parameter is configured to **0**, which is the default value, the query cache functionality is deactivated. If it is configured to 1, the functionality is executed in the primary query cache mode. If it is configured to 2, it is executed in the secondary query cache mode. To activate the query cache functionality, configure **max_plan_cache_entries**, **max_query_cache_entries** and **query_cache_mode** parameters equal to or greater than 1 respectively. Note that the query cache functionality is deactivated if any of these parameters does not satisfy the condition.

```
// The following is an example of caching up to 1,000 for query plans, caching up to 100
for query results.
max_plan_cache_entries=1000
max_query_cache_entries=100
query_cache_mode=1
// The configured values for the two parameters are invalid because the plan cache
functionality is deactivated.
max_plan_cache_entries=-1
max_query_cache_entries=100
query_cache_mode=1
// The plan cache functionality is executed for up to 1,000 query plans, and the query
cache functionality is deactivated.
max_plan_cache_entries=1000
```

```
max_query_cache_entries=100
query_cache_mode=0
```

query_cache_size_in_pages

query_cache_size_in_pages is a parameter that specifies the number of pages of query results to be cached. A query is cached only when its results are within the specified page size. If the parameter is configured to **-1**, which is the default value, the query cache functionality is executed for all queries without any constraints for the size of the result page.

Utility-Related Parameters

The following are parameters related to utilities used in CUBRID. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
compactdb_page_reclaim_only	int	0		
csql_history_num	int	50	1	200
communication_histogram	bool	no		
backup_volume_max_size_bytes	int	-1	1024*32	

compactdb_page_reclaim_only

compactdb_page_reclaim_only is a parameter related to the **compactdb** utility, which compacts the storage of already deleted objects to reuse OIDs of the already assigned storage. Storage optimization with the **compactdb** utility can be divided into three steps. The optimization steps can be selected through the **compactdb_page_reclaim_only** parameter. If the parameter is configured to **0**, which is the default value, step 1, 2 and 3 are all performed, so the storage is optimized in data, table and file units. If it is configured to 1, step 1 is skipped to have the storage optimized in table and file units. If it is configured to 2, steps 1 and 2 are skipped to have the storage optimized only in file units.

- Step 1 : Optimizes the storage only in data units.
- Step 2 : Optimizes the storage in table units.
- Step 3 : Optimizes the storage in file (heap file) units.

csql_history_num

csql_history_num is a parameter related to the CSQL Interpreter, and configures the number of SQL statements to be stored in the history of the CSQL Interpreter. The default value is **50**.

communication_histogram

communication_histogram is a parameter related to the **cubrid statdump** utility. It is related to Session Commands **;.h** of the CSQL Interpreter and the default value is **no**. For more information, see Outputting Statistics Information of Server.

backup_volume_max_size_bytes

backup_volume_max_size_bytes is a parameter that configures the size of the backup volume file created by the **cubrid backupdb** utility in byte units. If the parameter is configured to **-1**, which is the default value, the created backup volume is not partitioned; otherwise, the backup volume is partitioned as much as it is specified size.

there is no limit to the size of the backup volume to be created. If it is not configured, the size of the backup volume is allowed up to the size limit of the storage media.

HA-Related Parameters

The following are HA-related parameters. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value
ha_mode	string	off

ha_mode

The **ha_mode** parameter is used to set CUBRID HA features, and the default value is **off**.

- off : The CUBRID HA feature is not used.
- on : The CUBRID HA feature is used using the configured node as a node for failover.
- replica : The CUBRID HA feature is used without using the configured node as a node for failover.

To use the CUBRID HA feature, you should set HA related parameters in the **cubrid_ha.conf** file in addition to the **ha_mode** parameter. For more information, see CUBRID HA.

Other Parameters

The following are other parameters. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
service	string			
server	string			
index_scan_in_oid_order	bool	no		
single_byte_compare	bool	no		
insert_execution_mode	int	1	1	7
java_stored_procedure	bool	no		
pthread_scope_process	bool	yes		
auto_restart_server	bool	yes		
index_unfill_factor	float	0.05	0	0.5
use_orderby_sort_limit	bool	yes		
session_state_timeout	int	21600 (6 hours)	60 (1 min.)	31536000 (1 year)
multi_range_optimization_limit	int	100	0	100
access_ip_control	bool	no		
access_ip_control_file	string			

service

service is a parameter that registers a process that starts automatically when the CUBRID service starts. There are four types of processes: **server**, **broker**, **manager**, and **heartbeat**. Three processes are usually registered as in **service=server,broker,manager**.

- If the parameter is configured to **server**, the database process specified by the **@server** parameter gets started.
- If the parameter is configured to **broker**, the Broker process gets started.
- If the parameter is configured to **manager**, the manager process gets started.
- If the parameter is configured to **heartbeat**, the HA-related process gets started.

server

server is a parameter that registers a Database Server process that starts automatically when the CUBRID service starts.

index_scan_in_oid_order

index_scan_in_oid_order is a parameter that configures the result data to be retrieved in OID order after the index scan. If the parameter is configured to **no**, which is the default value, results are retrieved in data order; if it is configured to yes, they are retrieved in OID order.

single_byte_compare

single_byte_compare is a parameter that determines whether or not to compare strings in single byte units. If the parameter is configured to **no**, which is the default value, strings are compared in two byte units; if it is configured to yes, they are compared in single byte units. That is, you can retrieve/compare strings on data stored as UTF-8.

insert_execution_mode

insert_execution_mode has execution modes ranging from 1 to 7. Queries are usually executed on the server according to the query plan created by the client, but this parameter is used to directly insert queries on the server side. A selected execution mode is executed directly on the server, and other execution modes are executed on the client. This parameter can be used to perform an INSERT operation to the server in an environment in which dirty reading of INSERTed data is required, or in which the memory capacity of the client is limited.

The following are three types of **INSERT** statements for execution modes. This parameter can be set through a combination of integer values corresponding to each execution mode.

- **INSERT_SELECT** : When using the **SELECT** statement in the **INSERT** statement.

```
INSERT INTO code2(s_name, f_name) SELECT s_name, f_name from code;
```

- **INSERT_VALUES** : The common **INSERT** statement.

```
INSERT INTO code2(s_name, f_name) VALUES ('S', 'Silver');
```

- **INSERT_DEFAULT** : When inserting the default value because a column with the default value is omitted in the **INSERT** statement.

```
CREATE TABLE code2(s_name char(1) DEFAULT '_', f_name varchar(40));
```

```
INSERT INTO code2(f_name) DEFAULT VALUES;
```

- **INSERT_REPLACE** : For example, when the **REPLACE** statement is executed, the corresponding integer value is 8.

```
CREATE TABLE code2(s_name char(1) NOT NULL UNIQUE, f_name varchar(40));
REPLACE INTO code2 VALUES ('S', 'Silver');
```

- **INSERT_ON_DUP_KEY_UPDATE** : In addition, when the **ON DUPLICATE KEY UPDATE** clause is specified in the **INSERT** statement, the corresponding integer value is 16.

```
CREATE TABLE code2(s_name char(1) NOT NULL UNIQUE, f_name varchar(40));
INSERT INTO code2 VALUES ('S', 'Silver') ON DUPLICATE KEY UPDATE f_name='Silver';
```

The sum of the execution mode values above is the execution mode to be configured.

- Example 1 : If you want to execute **INSERT_SELECT** and **INSERT_VALUES** on the server, the **insert_execution_mode** is 3. (1 + 2 = 3)
- Example 2 : If you want to execute **INSERT_SELECT**, **INSERT_DEFAULT**, **INSERT_REPLACE**, an **INSERT_ON_DUP_KEY_UPDATE** on the server, the **insert_execution_mode** is 29(1+4+8+16=29).

java_stored_procedure

java_stored_procedure is a parameter that determines whether or not to use Java stored procedures by running the Java Virtual Machine (JVM). If the parameter is configured to **no**, which is the default value, JVM is not executed; if it is configured to yes, JVM is executed so you can use Java stored procedures. Therefore, configure the parameter to yes if you plan to use Java stored procedures.

pthread_scope_process

pthread_scope_process is a parameter that configures the contention scope of threads. It only applies to AIX systems. If the parameter is configured to no, the contention scope becomes **PTHREAD_SCOPE_SYSTEM**; if it is configured to yes, it becomes **PTHREAD_SCOPE_PROCESS**. The default value is **yes**.

auto_restart_server

auto_restart_server is a parameter that specifies whether or not to restart the process when it stops due to a fatal error in the Database Server process. If **auto_restart_server** is configured to yes, the server process restarts automatically when it stopped due to abnormal causes other than the normal stop process (**STOP** command of the CUBRID Server).

index_unfill_factor

If there is no free space because index pages are full when the **INSERT** or **UPDATE** operation is executed after the first index is created, the split of index page nodes occurs. This substantially affects the performance by increasing the operation time. **index_unfill_factor** is a parameter that specifies the percent of free space defined for each index page node when an index is created. The **index_unfill_factor** value is applied only when an index is created for the first time. The percent of free space defined for the page is not maintained dynamically. Its value ranges between 0 and 0.35. The default value is **0.05**.

If an index is created without any free space for the index page node (**index_unfill_factor**=0), the split of index page nodes occurs every time an additional insertion is made. This may degrade the performance.

If the value of **index_unfill_factor** is large, a large amount of free space is available when an index is created. Therefore, better performance can be obtained because the split of index nodes does not occur for a relatively long period of time until the free space for the nodes is filled after the first index is created.

If this value is small, the amount of free space for the nodes is small when an index is created. Therefore, it is likely that the index nodes are splitted by **INSERT** or **UPDATE** because the free space for the index nodes is filled in a short period of time.

use_orderby_sort_limit

The **use_orderby_sort_limit** parameter is used to determine whether or not to keep the intermediate result of sorting and merging process in the statement including the **ORDER BY … LIMIT** *row_count* clause as many as row_count. If it is set to yes, you can decrease unnecessary comparing and merging processes because as many as intermediate results will be kept as the value of *row_count*. The default value is yes.

session_state_timeout

The **session_state_timeout** parameter is used to define how long the CUBRID session data will be kept. The session data will be deleted when the driver terminates the connection or the session time is expired. The session time will expire if a client does not have any requests until the time specified by **session_state_timeout**.

The default value is **21600** seconds (6 hours).

The following are CUBRID seesion data.

- Custom variables defined by **SET**.
- **PREPARE** statements.
- **LAST_INSERT_ID**
- Number of records affected by the last executed statement(**ROW_COUNT**)

Custom variables defined by **SET** and **PREPARE** statements can be deleted by **DROP**/**DEALLOCATE** statements before session timeout.

multi_range_optimization_limit

If the number of rows specified by the **LIMIT** clause in the query, which has multiple ranges (col IN (?, ?, …,?)) and is available to use an index, is within the number specified by the **multi_range_optimization_limit** parameter, the optimization for the way of index sorting will be performed. The default value is 100.

For example, if a value for this parameter is set to 50, LIMIT 10 means that it is within the value specified by this parameter, so that the values that meet the conditions will be sorted to produce the result. If LIMIT is 60, it means that it exceeds the parameter configuration value, so that it gets and sorts out all values that meet the conditions.

Depending on the setting value, the differences are made between collecting the result with on-the-fly sorting of the intermediate values and sorting the result values after collecting them, and the bigger value could make more unfavorable performance.

access_ip_control

The **access_ip_control** parameter is used to limit the IP addresses available to access servers. The default value is no. For more information, see Database Server Access Limitation.

access_ip_control_file

The **access_ip_control_file** parameter is used to specify a file name in which the list of IPs allowed by servers is stored, if a value for **access_ip_control** is set to yes. For more information, see Database Server Access Limitation.

Changing Database Server Configuration

Editing the Configuration File

You can add/delete parameters or change parameter values by manually editing the system parameter configuration file (**cubrid.conf**) in the **$CUBRID/conf** directory.

The following parameter syntax rules are applied when configuring parameters in the configuration file:

- Parameter names are not case-sensitive.
- The name and value of a parameter must be entered in the same line.
- An equal sign (=) can be used to configure the parameter value. Spaces are allowed before and after the equal sign.
- If the value of a parameter is a character string, enter the character string without quotes. However, use quotes if spaces are included in the character string.

Using SQL Statements

Description

You can configure a parameter value by using SQL statements in the CSQL Interpreter or CUBRID Manager's Query Editor. Note that you cannot change every parameter. For updatable parameters, see cubrid.conf Configuration File and Default Parameters.

Syntax

```
SET SYSTEM PARAMETERS 'parameter_name=value [{; name=value}...]'
```

parameter_name is the name of a client parameter whose value is editable. In this syntax, value is the value of the given parameter. You can change multiple parameter values by separating them with semicolons (;). You must take caution when you apply changes of parameter values.

Example

The following is an example of retrieving the result of an index scan in OID order and configuring the number of queries to be saved in the history of the CSQL Interpreter to 70.

```
SET SYSTEM PARAMETERS 'index_scan_in_oid_order=1; csql_history_num=70'
```

Using Session Commands of the CSQL Interpreter

Description

You can configure system parameter values by using session commands (;**SEt**) in the CSQL Interpreter. Note that you cannot change every parameter. For updatable parameters, see cubrid.conf Configuration File and Default Parameters.

Example

The following is an example of configureing the **block_ddl_statement** parameter to 1 so that execution of DDL statements is not allowed.

```
csql> ;se block_ddl_statement=1
=== Set Param Input ===
block_ddl_statement=1
```

Broker Configuration

cubrid_broker.conf Configuration File and Default Parameters

Broker System Parameters

The following are Broker parameters that can be used in the **cubrid_broker.conf** configuration file. For description of each parameter, see **Parameter Description** in Parameter by Broker.

MAX_PREPARED_STMT_COUNT

int

2000 (min. : 1)

ACCESS_CONTROL

bool

no

ACCESS_CONTROL_FILE

string

Parameter Name	Type	Default Value
MASTER_SHM_ID	int	30001
ADMIN_LOG_FILE	string	log/broker/cubrid_broker.log
SERVICE	string	ON
BROKER_PORT	int	30000 (MAX : 65535)
MIN_NUM_APPL_SERVER	int	5
MAX_NUM_APPL_SERVER	int	40
APPL_SERVER_SHM_ID	int	30000
APPL_SERVER_MAX_SIZE	int	For Windows : 40 (32 bit), 80 (64 bit) For Linux : 0
LOG_DIR	string	log/broker/sql_log
ERROR_LOG_DIR	string	log/broker/error_log
SQL_LOG	string	ON
TIME_TO_KILL	int	120
SESSION_TIMEOUT	int	300
KEEP_CONNECTION	string	AUTO
ACCESS_LIST	string	-
ACCESS_LOG	string	ON
APPL_SERVER_PORT	int	BROKER_PORT+1
APPL_SERVER	string	CAS
LOG_BACKUP	string	OFF
SQL_LOG_MAX_SIZE	int	100000
MAX_STRING_LENGTH	int	-1

SOURCE_ENV	string	cubrid.env
STATEMENT_POOLING	string	ON
CCI_PCONNECT	string	OFF
SELECT_AUTO_COMMIT	string	OFF
LONG_QUERY_TIME	int	60
LONG_TRANSACTION_TIME	int	60
ACCESS_MODE	string	RW
CCI_DEFAULT_AUTOCOMMIT	string	OFF

Default Parameters

cubrid_broker.conf, a default broker configuration file created during the CUBRID installation, includes some default Broker parameters that must be changed. You can change the value of a parameter that is not included as a default parameter by manually adding or editing one.

The following is the content of the **cubrid_broker.conf** file that is created by default during the installation.

```
[broker]
MASTER_SHM_ID            =30001
ADMIN_LOG_FILE           =log/broker/cubrid_broker.log

[%query_editor]
SERVICE                  =ON
BROKER_PORT              =30000
MIN_NUM_APPL_SERVER      =5
MAX_NUM_APPL_SERVER      =40
APPL_SERVER_SHM_ID       =30000
LOG_DIR                  =log/broker/sql_log
ERROR_LOG_DIR            =log/broker/error_log
SQL_LOG                  =ON
TIME_TO_KILL             =120
SESSION_TIMEOUT          =300
KEEP_CONNECTION          =AUTO

[%BROKER1]
SERVICE                  =ON
BROKER_PORT              =33000
MIN_NUM_APPL_SERVER      =5
MAX_NUM_APPL_SERVER      =40
APPL_SERVER_SHM_ID       =33000
LOG_DIR                  =log/broker/sql_log
ERROR_LOG_DIR            =log/broker/error_log
SQL_LOG                  =ON
TIME_TO_KILL             =120
SESSION_TIMEOUT          =300
KEEP_CONNECTION          =AUTO
```

Environment Variables related to the Broker Configuration File

You can specify the **cubrid_broker.conf** file location by using the CUBRID_BROKER_CONF_FILE variable to executing various Brokers with different configuration.

Common Parameters

The following are parameters commonly applied to all Brokers, and they are listed under [broker] section in the **cubrid_broker.conf** file.

MASTER_SHM_ID

MASTER_SHM_ID is a parameter that specifies the identifier of shared memory which is used to manage the CUBRID Broker. Its value must be unique in the system. The default value is **30001**.

ADMIN_LOG_FILE

ADMIN_LOG_FILE is a parameter that specifies the file where the time information related with the CUBRID Broker running is stored. The default value is **log/broker/cubrid_broker.log** file.

Parameter by Broker

The following describes parameters to configure the environment variables of Brokers; each parameter is located under [%*broker_name*].

SERVICE

SERVICE is a parameter that determines whether to run the given Broker. It can be configured to either **ON** or **OFF**. The default value is **ON**. The Broker can run only when this parameter is configured to **ON**.

BROKER_PORT

BROKER_PORT is a parameter that configures the port number of the given Broker. Its value must be unique in the system and equal to or smaller than 65,535. By default, the broker port for **query_editor** is configured to **30000**, and the port for broker1 is configured to **33000**.

MIN_NUM_APPL_SERVER

MIN_NUM_APPL_SERVER is a parameter that configures the minimum number of application servers (CAS) even if any request to connect the broker has not been made. The default value is **5**.

MAX_NUM_APPL_SERVER

MAX_NUM_APPL_SERVER is a parameter that configures the maximum number of application servers (CAS). The default value is **40**. In an evnironment where connection pool is maintained by using a middleware such as WAS, you must specify the value of **MAX_NUM_APPL_SERVER** parameter as same as that of connection pool.

APPL_SERVER_SHM_ID

APPL_SERVER_SHM_ID is a parameter that configures the shared memory ID to be used by application servers (CAS). Its value must be unique in the system. The default value is the same as the port of the given Broker.

APPL_SERVER_MAX_SIZE

APPL_SERVER_MAX_SIZE is a parameter that specifies the maximum size of the process memory usage provided by the application server (CAS). The unit is MB. This value should be configured in the consideration of server operation environment because it affects the policy, CAS restart, in force. Especially, if you configure this value too low, applications can frequently be restarted. Note that the default value for Windows and Linux is different.
For Windows, the 32-bit CUBRID has 40 (MB) for the **APPL_SERVER_MAX_SIZE** value by default; 64-bit CUBRID has 80 (MB). If the current process memory usage exceeds the value of **APPL_SERVER_MAX_SIZE**, the Broker restarts the application server. For Linux, 0 is the default value for **APPL_SERVER_MAX_SIZE**; and it restarts the application server in the following conditions:

- Zero or negative : In case the current process is twice as large as the initial memory
- Positive : In case a value exceeds the number specified in **APPL_SERVER_MAX_SIZE**

LOG_DIR

LOG_DIR is a parameter that specifies the directory where SQL logs are stored. The default value is **log/broker/sql_log**. The file name of the SQL logs is *broker_name_id.sql***.log**.

ERROR_LOG_DIR

ERROR_LOG_DIR is a parameter that specifies the directory where error logs for the Broker are stored. The default value is **log/broker/error_log**. The name of the error log file for the Broker is *broker_ name_id***.err**.

SQL_LOG

SQL_LOG is a parameter that determines whether to leave logs for SQL statements processed by the application server (CAS) when an application server handles requests from a client. The default value is **ON**. When this parameter is configured to **ON**, all logs are stored. Log file name becomes *broker_name_id.sql***.log**. The file is created in the **log/broker/sql_log** directory under the installation directory. The parameter values are as follows:

- **OFF** : Does not leave any logs
- **ERROR** : Leaves logs for queries which occur an error. only queries where an error occurs
- **NOTICE** : Leaves logs for the long-duration execution queries which exceeds the configured time/transaction, or leaves logs for queries which occur an error
- **TIMEOUT** : Leaves logs for the long-duration execution queries which exceeds the configured time/transaction
- **ON**/**ALL** : Leaves all logs

TIME_TO_KILL

TIME_TO_KILL is a parameter that configures the time to remove application servers (CAS) in idle state among application servers added dynamically. The default value is **120** (sec). An idle state is one in which the server is not involved in any jobs. If this state continues exceeding the value specified in **TIME_TO_KILL**, the application server (CAS) is added or removed.

The value configured in this parameter affects only application server added dynamically, so it applies only when the **AUTO_ADD_APPL_SERVER** parameter is configured to **ON**. Note that times to add or remove the application servers (CAS) will be increased more if the **TIME_TO_KILL** value is so small.

SESSION_TIMEOUT

SESSION_TIMEOUT is a parameter that configures a timeout value for the session of the given Broker. The default value is **300** (sec). The given session is terminated if there is no response to the job request for the specified time period.

KEEP_CONNECTION

KEEP_CONNECTION is a parameter that specifies how application servers (CAS) and application clients are connected. It can be configured to **ON**, **OFF** or **AUTO**. If this parameter is configured to **OFF**, clients are connected to an application server in a transaction unit. If it is configured to **ON**, they are connected in a connection unit. If it is configured to **AUTO**, and then the number of application servers is more than that of clients, it will act as if ON; in the reverse case that clients are more than CASs, it will act as if OFF. The default value is **AUTO**.

ACCESS_LIST

ACCESS_LIST is a parameter that specifies the name of the file where IP addresses of application client which allows access of the CUBRID Broker is to be saved. To allow access by IP addresses 210.192.33.* and 210.194.34.*, save them to a file (ip_lists.txt) and then configure the file name with the value of this parameter.

ACCESS_LOG

ACCESS_LOG is a parameter that specifies whether or not to store access log. The default value is **ON**. The name of the access log file for the Broker is *broker_name_id***.access**, and the file is stored in the **$CUBRID/log/broker** directory.

LOG_BACKUP

LOG_BACKUP is a parameter that specifies whether or not to back up access and error log files of the Broker. The default value is **OFF**. If this parameter is configured to **ON**, access and error logs are backed up when the CUBRID Broker terminates. The backup file name for access logs becomes *broker_name_id* **access**, and the one for error logs becomes *broker_ name_id*. **error**.

SQL_LOG_MAX_SIZE

SQL_LOG_MAX_SIZE is a parameter that specifies the maximum size of the SQL log file. The default value is **100,000** (KB). If the size of the SQL log file, which is created when the **SQL_LOG** parameter is configured to **ON**, reaches the value configured by the parameter, *broker_name_id*. **sql.log.bak** is created.

APPL_SERVER_PORT

APPL_SERVER_PORT, which can be added only in the Windows operating system, is a parameter that specifies the connection port for the application server (CAS) which communicates with the application client. The default is configured to add 1 to the specified **BROKER_PORT** parameter. As the maximum number of application servers is limited by the **MAX_NUM_APPL_SERVER** parameter of the **cubrid_broker_conf** file, the maximum number of connection ports for the application server (CAS) is also limited by the value of the **MAX_NUM_APPL_SERVER** parameter. If there is a firewall in the Windows operating system between application client and the CUBRID Broker, the connection port specified by **BROKER_PORT** and **APPL_SERVER_PORT** must be open.

APPL_SERVER

APPL_SERVER is a parameter that specifies the type of application servers created and managed by the CUBRID Broker. The default value is **CAS**.

MAX_STRING_LENGTH

MAX_STRING_LENGTH is a parameter that configures the maximum string length for bit, varbit, char, varchar, nchar, nchar varying data types. If this parameter is configured to **-1**, which is the default value, the length defined in the database is used. If the parameter is configured to **100**, the value 100 is applied even when a certain attribute is defined as varchar(1000).

SOURCE_ENV

SOURCE_ENV is a parameter that specifies the file to independently configure operating system environment variables for each broker. The extension of the file must be **env**. All parameters specified in **cubrid.conf** can also be configured by environment variables. For example, the **lock_timeout_in_secs** parameter in **cubrid.conf** can also be configured by the **CUBRID_LOCK_TIMEOUT_IN_SECS** environment variable. As another example, to block execution of DDL statements on broker1, you can configure **CUBRID_BLOCK_DDL_STATEMENT 1** in the file specified by **SOURCE_ENV**.

An environment variable, if exists, has priority over **cubrid.conf**.

The default value is **cubrid.env**.

STATEMENT_POOLING

STATEMENT_POOLING is a parameter that specifies whether or not to use statement pooling. The default value is **ON**.

When a transaction is committed or rolled back, CUBRID closes all the prepared statement handles that exist in the client session. However, if the parameter is set to **STATEMENT_POOLING=ON**, the prepared statement handles remain in the pool, so that the handles can be reused. Therefore, you must maintain the default setting (*STATEMENT_POOLING=ON*) in general applications that reuse prepared statements or in environments in which a library such as DBCP, in which the statement pooling is implemented, is applied.

When the parameter is set to **STATEMENT_POOLING=OFF** and the prepared statement is executed after the transaction is committed or terminated, the following message is displayed.

```
Caused by: cubrid.jdbc.driver.CUBRIDException: Attempt to access a closed Statement.
```

CCI_PCONNECT

CCI_PCONNECT is a parameter that specifies whether or not to use the CCI connection pooling. The default value is **OFF**. This parameter affects the applications using CCI API or interfaces such as PHP, ODBC, or OLE DB, which are developed in CCI API.

SELECT_AUTO_COMMIT

SELECT_AUTO_COMMIT is a parameter that sets auto-commit mode for **SELECT** statements in **CCI** or **PHP**. Its default value is **OFF**. However, note that auto-commit is performed only at the point at which the result set for all n query statements is fetched from the server when there are n prepared statements. An example is as follows. For more information, see cci_end_tran.

```
SELECT_1 prepare
SELECT_1 execute // AUTO COMMIT O

SELECT_1 prepare
SELECT_2 prepare
SELECT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT_2 execute // AUTO COMMIT O

SELECT_1 prepare
SELECT_1 execute // AUTO COMMIT O
INSERT_1 prepare
INSERT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

INSERT_1 prepare
INSERT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT_1 prepare
SELECT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

SELECT_1 prepare
INSERT_1 prepare
SELECT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
INSERT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

INSERT_1 prepare
SELECT_1 prepare
INSERT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
```

LONG_QUERY_TIME

LONG_QUERY_TIME is a parameter that specifies execution time which is evaluated as long-duration queries. The default value is **60000** in ms. Note that a parameter value is configured to 0, it is not evaluated as a long-duration query.

LONG_TRANSACTION_TIME

LONG_TRANSACTION_TIME is a parameter that specifies execution time which is evaluated as long-duration transactions. The default value is **60000** in ms. Note that a parameter is configured to 0, it is not evaluated as a long-duration transaction.

CCI_DEFAULT_AUTO_COMMIT

CCI_DEFAULT_AUTO_COMMIT is a parameter that specifies the automatic commit of applications developed by CCI API. The default value is **OFF**.

This parameter affects the applications developed by CCI API or the applications using interfaces (PHP, ODBC, and OLE DB) developed by CCI; it does not affect the applications developed by JDBC.

MAX_PREPARED_STMT_COUNT

MAX_PREPARED_STMT_COUNT is a parameter that limits the number of prepared statements by user (application) access. The default value is **2000** and the minimum value is 1. By making user specify the parameter value, creation of prepared statement exceeding memory allowed by system can be prohibited.

ACCESS_CONTROL

ACCESS_CONTROL is a parameter that limits the application that access a broker. The default value is **OFF**. For more information, see Broker Server Access Limitation.

ACCESS_CONTROL_FILE

ACCESS_CONTROL_FILE is a parameter that specifies database names, database user IDs, and file names including the list of IPs, which are allowed to access a broker. For more information, see Broker Server Access Limitation.

API Reference

API Reference

This chapter covers the following APIs:

- JDBC API
- ODBC API
- OLE DB API
- PHP API
- CCI API

JDBC API

JDBC Programming

CUBRID JDBC Driver

The CUBRID JDBC driver (**cubrid_jdbc.jar**) enables the system to make a connection to the CUBRID database in an application written in Java. The driver is located in the "location of CUBRID installed/jdbc" directory.

The CUBRID JDBC driver has been developed based on the JDBC 2.0 specification and provides compilation output generated in JDK version 1.6.

Checking the CUBRID JDBC Driver Version

You can check the JDBC driver version as follows:

```
% jar -tf cubrid_jdbc.jar
META-INF/ META-INF/MANIFEST.MF
cubrid/ cubrid/jdbc/
cubrid/jdbc/driver/
cubrid/jdbc/jci/
cubrid/sql/
CUBRID-JDBC-8.1.4.1032
cubrid/jdbc/driver/CUBRIDBlob.class
...
```

Registering the CUBRID JDBC Driver

Use the **Class.forName** (*driver-class-name*) command to register the JDBC driver. The following is an example of loading the cubrid.jdbc.driver.CUBRIDDriver class to register the CUBRID JDBC driver.

```
import java.sql.*;
import cubrid.jdbc.driver.*;

public class LoadDriver {
   public static void main(String[] Args) {
       try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       } catch (Exception e) {
           System.err.println("Unable to load driver.");
           e.printStackTrace();
       }
       ...
```

CUBRID JDBC Interface

The following table shows the JDBC standard and extended interfaces supported by CUBRID JDBC. Note that some methods are not supported even though they are specified in the JDBC 2.0 specification.

Supported Inferface by CUBRID

JDBC Standard Interface	JDBC Extended Interface	Supported
java.sql.Blob java.sql.CallableStatement java.sql.Clob java.sql.Connection java.sql.DatabaseMetaData java.sql.Driver java.sql.PreparedStatement java.sql.ResultSet java.sql.ResultSetMetaData	java.sql.CUBRIDConnection java.sql.CUBRIDPreparedStatement java.sql.CUBRIDResultSet java.sql.CUBRIDResultSetMetaData CUBRIDOID	Supported.
java.sql.Statement	java.sql.CUBRIDStatement	The getGeneratedKeys()

		method of JDBC 3.0 is supported.
java.sql.DriverManager		Supported
Java.sql.SQLException	Java.sql.CUBRIDException	Supported
java.sql.Array java.sql.ParameterMetaData java.sql.Refava.sql.Savepoint java.sql.SQLData java.sql.SQLInput java.sql.Struct		Not Supported

Connection Configuration

The **DriverManager** is a basic interface for JDBC driver management and performs functions such as selecting a database driver and creating a new database connection. If the CUBRID JDBC driver is registered, database connection is made by calling the **DriverManager.getConnection** (*db-url*, *user-id*, *password*) function. The **getConnection** function returns the **Connection** object, which is used for query and command executions and transaction commit or rollback. The parameter *db-url*, which is for connection configuration, is as follows:

```
jdbc:cubrid:<host>:<port>:<db-name>:[user-id]:[password]:[?<property> [& <property>]]

<host> ::=
hostname | ip_address

<property> ::=
althosts= <alternative_hosts> | rctime= <second> | charset= <character_set>

<alternative_hosts> :
<standby_broker1_host>:<port> [,<standby_broker2_host>:<port>]
```

- *<host>* : IP address or host name where the CUBRID Broker is running
- *<port>* : Broker port number (default : 33000)
- *<db-name>* : The name of the database to connect
- *[user-id]* : The user that will be connected to the database. There are two users in the database by default: DBA and PUBLIC. If you enter an empty string (" "), you will connect to the database as a PUBLIC user.
- *[password]* : If there is no password set for the user, enter an empty string (" ").
- *althosts* : One or more host IP of standby broker and connection port to be failed over in HA environment
- *rctime* : Interval time (in seconds) to fail over an active server during system failure. For more information, see the example in the HA-Related JDBC Configuration.
- *charset* : Character set (charset) of database to be connected

Example 1

```
--connection URL string when user name and password omitted

URL=jdbc:CUBRID:127.0.0.1:31000:db1:::

--connection URL string when charset property specified

URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?charset=utf-8

--connection URL string when a property(althosts) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000

--connection URL string when properties(althosts,rctime) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600

--connection URL string when properties(althosts,rctime, charset) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600
&charset=utf-8
```

Example 2

```
String url = "jdbc:cubrid:210.216.33.250:43300:demodb:::";
String userid = "";
String password = "";

try {
   Connection conn =
           DriverManager.getConnection(url,userid,password);
   // Do something with the Connection

   ...

   } catch (SQLException e) {
       System.out.println("SQLException:" + e.getMessage());
       System.out.println("SQLState: " + e.getSQLState());
   }
   ...
```

Note The rollback function, which requests the transaction rollback, exits when the server completes the work.

Verifying Foreign Key Information

Description

You can verify foreign key information by using **getImportedKeys**, **getExportedKeys**, and **getCrossReference** methods provided by **DatabaseMetaData** interface. Usage and examples of each method are as follows:

Syntax

```
getImportedKeys(String catalog, String schema, String table)
getExportedKeys(String catalog, String schema, String table)
getCrossReference(String parentCatalog, String parentSchema, String parentTable, String
foreignCatalog, String foreignSchema, String foreignTable)
```

- **getImportedKeys method** : A method that retrieves the information of primary key columns which are referred by foreign key columns in a given table. The results are sorted by **PKTABLE_NAME** and **KEY_SEQ**.
- **getExportedKeys method** : A method that retrieves the information of all foreign key columns which refer to primary key columns in a given table. The results are sorted by **FKTABLE_NAME** and **KEY_SEQ**.
- **getCrossReference method** : A method that retroeves the information of primary key columns which are referred by foreign key columns in a given table. The results are sorted by **PKTABLE_NAME** and **KEY_SEQ**.

Return Value

When the methods above are called, the following ResultSet, consisting of 14 columns, is returned.

Name	Type	Note
PKTABLE_CAT	String	Always null
PKTABLE_SCHEM	String	Always null
PKTABLE_NAME	String	Table name of primary key
PKCOLUMN_NAME	String	Table name of primary key
FKTABLE_CAT	String	Always null
FKTABLE_SCHEM	String	Always null
FKTABLE_NAME	String	Table name of foreign key
FKCOLUMN_NAME	String	Column name of foreign key
KEY_SEQ	short	Sequence of foreign or primary keys (starting from 1)
UPDATE_RULE	short	A corresponding value to referring action defined as to foreign keys when primary keys are updated Cascade=0, Restrict=2, No action=3, Set null=4

DELETE_RULE	short	A corresponding value to referring action defined as to foreign keys when primary keys are deleted Cascade=0, Restrict=2, No action=3, Set null=4
FK_NAME	String	Foreign key name
PK_NAME	String	Primary key name
DEFERRABILITY	short	Always 6(DatabaseMetaData.importedKeyInitiallyImmediate)

Example

```
ResultSet rs = null;

                DatabaseMetaData dbmd = conn.getMetaData();

                System.out.println("\n===== Test getImportedKeys");
                System.out.println("=====");
                rs = dbmd.getImportedKeys(null, null, "pk_table");
                Test.printFkInfo(rs);
                rs.close();

                System.out.println("\n===== Test getExportedKeys");
                System.out.println("=====");
               rs = dbmd.getExportedKeys(null, null, "fk_table");
                Test.printFkInfo(rs);
                rs.close();

                System.out.println("\n===== Test getCrossReference");
                System.out.println("=====");
                rs = dbmd.getCrossReference(null, null, "pk_table", null, null,
"fk_table");

                Test.printFkInfo(rs);
                rs.close();
```

Using OIDs and Collections

In addition to the methods defined in the JDBC specification, the CUBRID JDBC driver provides methods that handle OIDs and collections (set, multiset and sequence).

To use these methods, you must import **cubrid.sql.*;** in addition to the CUBRID JDBC driver classes which are imported by default. In addition, to get the results, you must convert **ResultSet** to **CUBRIDResultSet** first. (**ResultSet** is provided by the standard JDBC API, by default.)

```
import cubrid.jdbc.driver.* ;
import cubrid.sql.* ;
...
CUBRIDResultSet urs = (CUBRIDResultSet) stmt.executeQuery(
"SELECT city FROM location");
```

Caution **AUTO COMMIT** does not work even though it is configured to **TRUE** if CUBRID extended APIs are used. Therefore, you must manually commit open connections. The CUBRID extended APIs are methods that handle OIDs and collections.

Using OIDs

You must follow the following rules to use OIDs.

- To use **CUBRIDOID**, you should import **cubrid.sql.*;**. (a)
- You can retrieve an OID by specifying a class name in the **SELECT** statement. The name can be used together with other attributes. (b)
- The **ResultSet** of a query must be **CUBRIDResultSet**. (c)
- The method that retrieves the OID from the **CUBRIDResultSet** is **getOID**(). (d)
- To retrieve a value from an OID, use the **getValues**() method. Its result is **ResultSet**. (e)
- To substitute a value for an OID, use the **setValues**() method. (f)
- When you use the extended APIs, you must always perform **commit**() to make connection. (g)

```
import java.sql.*;
import cubrid.sql.*; //a
import cubrid.jdbc.driver.*;

/*
CREATE TABLE oid_test(
   id INTEGER,
   name VARCHAR(10),
   age INTEGER
);

INSERT INTO oid_test VALUES(1, 'Laura', 32);
INSERT INTO oid_test VALUES(2, 'Daniel', 39);
INSERT INTO oid_test VALUES(3, 'Stephen', 38);
*/

class OID_Sample
{
   public static void main (String args [])
   {
      // Making a connection
      String url= "jdbc:cubrid:localhost:33000:demodb:::";
      String user = "dba";
      String passwd = "";

      // SQL statement to get OID values
      String sql = "SELECT oid_test from oid_test"; //b
      // columns of the table
      String[] attr = { "id", "name", "age" } ;


      // Declaring variables for Connection and Statement
      Connection con = null;
      Statement stmt = null;
      CUBRIDResultSet rs = null;
      ResultSetMetaData rsmd = null;

      try {
         Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
      } catch (ClassNotFoundException e) {
         throw new IllegalStateException("Unable to load Cubrid driver", e);
      }

      try {
         con = DriverManager.getConnection(url, user, passwd);
         stmt = con.createStatement();
         rs = (CUBRIDResultSet)stmt.executeQuery(sql); //c
         rsmd = rs.getMetaData();

         // Printing columns
         int numOfColumn = rsmd.getColumnCount();
         for (int i = 1; i <= numOfColumn; i++ ) {
            String ColumnName = rsmd.getColumnName(i);
            String JdbcType = rsmd.getColumnTypeName(i);
            System.out.print(ColumnName );
            System.out.print("("+ JdbcType + ")");
            System.out.print(" | ");
         }
         System.out.print("\n");
         // Printing rows
         CUBRIDResultSet rsoid = null;
         int k = 1;

         while (rs.next()) {
            CUBRIDOID oid = rs.getOID(1); //d
            System.out.print("OID");
            System.out.print(" | ");
            rsoid = (CUBRIDResultSet)oid.getValues(attr); //e

            while (rsoid.next()) {
               for( int j=1; j <= attr.length; j++ ) {
                  System.out.print(rsoid.getObject(j));
                  System.out.print(" | ");
```

```
                }
            }
            System.out.print("\n");

            // New values of the first row
            Object[] value = { 4, "Yu-ri", 19 };
            if (k == 1) oid.setValues(attr, value); //f

            k = 0;
         }
         con.commit(); //g

      } catch(CUBRIDException e) {
         e.printStackTrace();

      } catch(SQLException ex) {
         ex.printStackTrace();

      } finally {
         if(rs != null) try { rs.close(); } catch(SQLException e) {}
         if(stmt != null) try { stmt.close(); } catch(SQLException e) {}
         if(con != null) try { con.close(); } catch(SQLException e) {}
      }
   }
}
```

Using Collections

The line marked by 'a' in the example 1 below is where data of a collection type (SET, MULTISET, LIST) is fetched from the **CUBRIDResultSet**. The results are returned as array format. Note that this function is supported only when data types of elements defined in the collection type are same.

Example 1

```
import java.sql.*;
import java.lang.*;
import cubrid.sql.*;
import cubrid.jdbc.driver.*;

// create class collection_test(
// settest set(integer),
// multisettest multiset(integer),
// listtest list(Integer)
// );
//

// insert into collection_test values({1,2,3},{1,2,3},{1,2,3});
// insert into collection_test values({2,3,4},{2,3,4},{2,3,4});
// insert into collection_test values({3,4,5},{3,4,5},{3,4,5});

class Collection_Sample
{
   public static void main (String args [])
   {
       String url= "jdbc:cubrid:210.216.33.250:43300:demodb:::";
       String user = "";
       String passwd = "";
       String sql = "select settest,multisettest,listtest from collection_test";
       try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       } catch(Exception e){
           e.printStackTrace();
       }
       try {
           Connection con = DriverManager.getConnection(url,user,passwd);
           Statement stmt = con.createStatement();
           CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);
           CUBRIDResultSetMetaData rsmd = (CUBRIDResultSetMetaData) rs.getMeta Data();
           int numbOfColumn = rsmd.getColumnCount();
           while (rs.next ()) {
               for (int j=1; j<=numbOfColumn; j++ ) {
                   Object[] reset = (Object[]) rs.getCollection(j); //a
```

```
                   for (int m=0 ; m < reset.length ; m++)
                       System.out.print(reset[m] +",");
                   System.out.print(" | ");
               }
               System.out.print("\n");
           }
           rs.close();
           stmt.close();
           con.close();
       } catch(SQLException e) {
           e.printStackTrace();
       }
   }
}
```

Example 2

```
import java.sql.*;
import java.io.*;
import java.lang.*;
import cubrid.sql.*;
import cubrid.jdbc.driver.*;

// create class collection_test(
// settest set(integer),
// multisettest multiset(integer),
// listtest list(Integer)
// );
//
// insert into collection_test values({1,2,3},{1,2,3},{1,2,3});
// insert into collection_test values({2,3,4},{2,3,4},{2,3,4});
// insert into collection_test values({3,4,5},{3,4,5},{3,4,5});

class SetOP_Sample
{
   public static void main (String args [])
   {
       String url = "jdbc:cubrid:127.0.0.1:33000:demodb:::" ;
       String user = "";
       String passwd = "";
       String sql = "select collection_test from collection_test";
       try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       } catch(Exception e){
           e.printStackTrace();
       }
       try {
           CUBRIDConnection con =(CUBRIDConnection)
           DriverManager.getConnection(url,user,passwd);
           Statement stmt = con.createStatement();
           CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);
           while (rs.next ()) {
               CUBRIDOID oid = rs.getOID(1);
               oid.addToSet("settest",new Integer(10));
               oid.addToSet("multisettest",new Integer(20));
               oid.addToSequence("listtest",1,new Integer(30));
               oid.addToSequence("listtest",100,new Integer(100));
               oid.putIntoSequence("listtest",99,new Integer(99));
               oid.removeFromSet("settest",new Integer(1));
               oid.removeFromSet("multisettest",new Integer(2));
               oid.removeFromSequence("listtest",99);
               oid.removeFromSequence("listtest",1);
           }
           con.commit();
           rs.close();
           stmt.close();
           con.close();
       } catch(SQLException e) {
           e.printStackTrace();
       }
   }
}
```

Getting Auto-Increment Column Values

Auto-increment Feature

The auto-increment feature (**AUTO_INCREMENT**) is a column-related feature that increments the numeric value of each row. For more information, see Column Definition in Creating Tables. This feature can be defined only for numeric domains (**SMALLINT**, **INTEGER**, **DECIMAL**(*p*, 0), **NUMERIC**(*p*, 0)).

The auto-increment feature is recognized as an automatically created key in a JDBC program. To retrieve the key, you need to specify the time to insert a row from which the automatically created key value is to be retrieved. To perform it, you must set the flag by calling **Connection.prepareStatement** and **Statement.execute**. In this case, the command to be executed should be the **INSERT** statement or **INSERT** within **SELECT** statement. For other commands, the JDBC driver ignores the flag-setting parameter.

Steps

- Use one of the followings to indicate whether or not to return a key created automatically. The following method forms are used for tables of the database server that supports the auto-increment columns. Each method form can be applied only to a single-row **INSERT** statement.
- Create a **PreparedStatement** object by referring to the followings:
  Connection.prepareStatement(*sql statement*, **Statement.RETURN_GENERATED_KEYS**);
- To insert a row using the **Statement.execute** method, use one of the forms of the **Statement.execute** method by referring to the followings:
  Statement.execute(*sql statement*, **Statement.RETURN_GENERATED_KEYS**);
- Retrieve a **ResultSet** object that contains a automatically created key value by calling the **PreparedStatement.getGeneratedKeys** or **Statement.getGeneratedKeys** method. Note that the data type of the automatically created key in **ResultSet** is **DECIMAL** regardless of the data type of the given domain.

Example

The following is an example of creating a table with the auto-increment feature, entering data into the table so that automatically created key values are entered into auto-increment columns, and checking whether the key values are successfully retrieved by using the **Statement.getGeneratedKeys**() method. Each step is explained in the comments for commands that correspond to the steps above.

```
import java.sql.*;
import java.math.*;
import cubrid.jdbc.driver.*;

Connection con;
Statement stmt;
ResultSet rs;
java.math.BigDecimal iDColVar;
...
stmt = con.createStatement();     // Create a Statement object

stmt.executeUpdate(
"CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), "
+   "IDENTCOL INTEGER AUTO_INCREMENT)"); // Create table with identity column

stmt.execute(
"INSERT INTO EMP_PHONE (EMPNO, PHONENO) "
+   "VALUES ('000010', '5555')",          // Insert a row  <Step 1>
Statement.RETURN_GENERATED_KEYS);            // Indicate you want automatically


rs = stmt.getGeneratedKeys();    // generated keys
                                       // Retrieve the automatically  <Step 2>
                                       // generated key value in a ResultSet.
                                       // Only one row is returned.
                                       // Create ResultSet for query
while (rs.next()) {
  java.math.BigDecimal idColVar = rs.getBigDecimal(1);
                                       // Get automatically generated key
                                       // value
  System.out.println("automatically generated key value = " + idColVar);
```

```
}
rs.close();                             // Close ResultSet
stmt.close();                           // Close Statement
```

Using BLOB/CLOB

The interfaces that porcess **LOB** data in JDBC is implemented based on JDBC 4.0 specification. The constraints of interfaces are as follows:

- It supports sequential writes only when creating the objects of **BLOB** or **CLOB**. Writing to arbitary locations are not supported.
- You cannot change the data of **BLOB** or **CLOB** by calling methods of **BLOB** or **CLOB** object which are received from **ResultSet**.
- It does not support **Blob.truncate**, **Clob.truncate**, **Blob.position**, and **Clob.position**.
- You cannot bind **LOB** data by calling **PreapredStatement.setAsciiStream**, **PreparedStatement.setBinaryStream**, and **PreparedStatement.setCharacterStream** methods of **BLOB**/**CLOB** type columns.
- To use **BLOB**/**CLOB** types in an environment where JDBC 4.0 specification is not supported such as JDB version 1.5 or earlier, you must convert a conn object to **CUBRIDConnection**, explicitly. See the example below.

```
// JDK 1.6 or later
import java.sql.*;
Connection conn = DriverManager.getConnection(url, id, passwd);
Blob blob = conn.createBlob();
…
// JDK 1.5 or earlier
import java.sql.*;
import cubrid.jdbc.driver.*;

Connection conn = DriverManager.getConnection(url, id, passwd);
Blob blob = ((CUBRIDConnection)conn).createBlob();
…
```

Saving LOB Data

The way to bind **LOB** type data is as follows:

- Create **java.sql.Blob** or **java.sql.Clob** object and save the file contents in the object. Use, then, **setBlob**() or **setClob**() of **PreparedStatement** (example 1).
- Perform query and get **java.sql.Blob** or **java.sql.Clob** object from the **ResultSet object**. Bind, then, the object in **PreparedStatement** (example 2).

Example 1

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
Connection conn = DriverManager.getConnection ("jdbc:cubrid:localhost:33000:image_db:::",
"", "");
PreparedStatement pstmt1 = conn.prepareStatement("INSERT INTO doc(image_id, doc_id, image)
VALUES (?,?,?)");
pstmt1.setString(1, "image-21");
pstmt1.setString(2, "doc-21");

//Creating an empty file in the file system
Blob bImage = conn.createBlob();
byte[] bArray = new byte[256];
…

//Inserting data into the external file. Position is start with 1.
bImage.setBytes(1, bArray);
//appending data into the external file
bImage.setBytes(257, bArray);
…
pstmt1.setBlob(3, bImage);
pstmt1.executeUpdate();
…
```

Example 2

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
```

```
Connection conn = DriverManager.getConnection ("jdbc:cubrid:localhost:33000:image_db:::",
"", "");
conn.setAutoCommit(false);
PreparedStatement pstmt1 = conn.prepareStatement("SELECT image FROM doc WHERE image_id = ?
");
pstmt1.setString(1, "image-21");
ResultSet rs = pstmt1.executeQuery();

while (rs.next())
{
Blob bImage = rs.getBlob(1);
PreparedStatement pstmt2 = conn.prepareStatement("INSERT INTO doc(image_id, doc_id, image)
VALUES (?,?,?)");
pstmt2.setString(1, "image-22")
pstmt2.setString(2, "doc-22")
pstmt2.setBlob(3, bImage);
pstmt2.executeUpdate();
pstmt2.close();
}
pstmt1.close();
conn.commit();
conn.setAutoCommit(true);
conn.close();
…
```

Getting LOB Data

The way to get **LOB** type data is as follows:

- Get data directly from **ResultSet** by using **getBytes**() or **getString**() method (example 1).
- Get the **java.sql.Clob** object from **ResultSet** by calling **getBlob**() or **getClob**() method and then get data by using **getBytes**() or **getSubString**() method for this object (example 2).

Example 1

```
Connection conn = DriverManager.getConnection ("jdbc:cubrid:localhost:33000:image_db:::",
"", "");

// Getting data directly from ResetSet
PrepareStatement pstmt1 = conn.prepareStatement("SELECT content FROM doc_t WHERE doc_id = ?
");
pstmt2.setString(1, "doc-10");
ResultSet rs = pstmt1.executeQuery();
while (rs.next())
  {
    String sContent = rs.getString(1);
    System.out.println("doc.content= "+sContent.);
  }
```

Example 2

```
Connection conn = DriverManager.getConnection ("jdbc:cubrid:localhost:33000:image_db:::",
"", "");

//Getting Blob data from ResultSet and getting data from the Blob object
PrepareStatement pstmt2 = conn.prepareStatement("SELECT image FROM image_t WHERE image_id
= ?");
pstmt2.setString(1,"image-20");
ResultSet rs = pstmt2.executeQuery();
while (rs.next())
  {
    Blob bImage = rs.getBlob(1);
    Bytes[] bArray = bImange.getBytes(1, (int)bImage.length());
  }
```

Note If a string longer than defined size in a column is inserted(INSERT) or updated(UPDATE), the string will be truncated.

CUBRIDOID

Overview

A **CUBRIDOID** class contains the following methods to process OIDs.

Return Type	Method Name
void	addToSequence(String attrName, int index, Object value)
void	addToSet(String attrName, Object value)
static CUBRIDOID	getNewInstance(CUBRIDConnection con, String oidStr)
String	getOidString()
String	getTableName()
ResultSet	getValues(String[] attrNames)
Boolean	isInstance()
void	putIntoSequence(String attrName, int index, Object value)
void	remove()
void	removeFromSequence(String attrName, int index)
void	removeFromSet(String attrName, Object value)
void	setReadLock()
void	setValues(String[] attrNames, Object[] values)
void	setWriteLock()

addToSequence

Description

This function is used to insert the value specified in *value* into the attribute named *attrName* and associated with **SEQUENCE** constraints on the **CUBRIDOID** instance, specifically in front of the *index*-th element in the **SEQUENCE** attribute.

Syntax

```
void addToSequence(String attrName, int index, Object value)
```

Example

```
//create class foo(c list of int  )
//insert into foo values({3})

String sql = "select foo from foo" ;

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);               // get OID
   oid.addToSequence("c",1, new Integer(22)); // c: {3}-> {22,3}
}
```

addToSet

Description

This function is used to insert the value specified in *value* into the attribute named *attrName* and associated with **SET** or **MULTISET** constraints on the **CUBRIDOID** instance.

Syntax

```
void addToSet(String attrName, Object value)
```

Example

```
//create class foo(a set of int, b multiset of int )
//insert into foo values({1},{2})
String sql = "select foo from foo" ;

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);
while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);        // get OID
   oid.addToSet("a",new Integer(11));  // a : {1} -> {1,11}
   oid.addToSet("b",new Integer(13));  // b : {2} -> {2, 13}
}
```

getNewInstance

Description

This function is used to convert an OID string to a **CUBRIDOID** object, and then returns the **CUBRIDOID** object.

Syntax

```
static CUBRIDOID getNewInstance(CUBRIDConnection con, String oidStr)
```

Return Value

- **CUBRIDOID** object

Example

```
String sql = "select foo from foo" ;

CUBRIDConnection con = (CUBRIDConnection)
                    DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID realoid = rs.getOID(1);  // get OID (CUBRIDOID)
   // CUBRIDOID -> OID string
   String stringoid =  realoid.getOidString();
   // OID string -> CUBRIDOID
   realoid = CUBRIDOID.getNewInstance(con, stringoid);
}
```

getOidString

Description

This function is used to convert a **CUBRIDOID** object to an OID string, and then returns the string.

Syntax

```
String getOidString()
```

Return Value

- Character string

Example

```
String sql = "select foo from foo" ;

CUBRIDConnection con = (CUBRIDConnection)
                    DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID realoid = rs.getOID(1);    // get OID
   // CUBRIDOID -> OID string
   String stringoid = realoid.getOidString();
   // OID string -> CUBRIDOID
   realoid = CUBRIDOID.getNewInstance(con,stringoid);
}
```

getTableName

Description

This function is used to returns the table name of the instance corresponding to the **CUBRIDOID** object.

Syntax

```
String getTableName()
```

Return Value

- A table name of an instance that corresponds to **CUBRIDOID**

Example

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);

   String tablename = oid.getTableName();
   System.out.println(tablename );
}
```

getValues

Description

This function is used to return the **ResultSet** which contains values of the requested attribute.

Syntax

```
ResultSet getValues(String[] attrNames)
```

Return Value

- **ResultSet**

Example

```
// create class foo ( a string, b int )
// insert into foo values('CUBRID', 2001)

String sql = "select foo from foo";
String[] attr = { "a", "b" };  // class's column name list
CUBRIDResultSet rs= (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   ResultSet rsoid = oid.getValues(attr);
}
```

isInstance

Description

This function is used to return true if the instance corresponding to the **CUBRIDOID** exists. If otherwise, it returns false.

Syntax

```
Boolean isInstance()
```

Return Value

- TRUE : An instance that corresponds to **CUBRIDOID** exists.
- FALSE : An instance that corresponds to **CUBRIDOID** does not exist.

Example

```
String sql = "select foo  from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   System.out.print("isInstance : " + oid.isInstance()); // true
   oid.remove();  // remove the object in the oid
   System.out.print("After remove, isInstance : "
                    + oid.isInstance()); // false
}
```

putIntoSequence

Description

This function is used to modify the *index*-th value in the attribute associated with the **SEQUENCE** constraint on the **CUBRIDOID** instance as the value specified in *value*.

Syntax

```
void putIntoSequence(String attrName, int index, Object value)
```

Example

```
//create class foo(c list of int  )
//insert into foo values({1})

String sql = "select foo from foo" ;

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
```

```
   CUBRIDOID oid = rs.getOID(1);                // get OID
   oid.putIntoSequence("c",1, new Integer(10)); // c:{1}->{10}
}
```

remove

Description

This function is used to remove the instance corresponding to the **CUBRIDOID**.

Syntax

```
void remove()
```

Example

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   System.out.print("isInstance : " + oid.isInstance()); // true
   oid.remove();  // remove the object in the oid
   System.out.print("  After remove .isInstance : " +
                    oid.isInstance()); // false
}
```

removeFromSequence

Description

This function is used to remove the *index*-th value from the attribute associated with the **SEQUENCE** constraint on the **CUBRIDOID** instance.

Syntax

```
void removeFromSequence(String attrName, int index)
```

Example

```
//create class foo(c list of int  )
//insert into foo values(1,3)

String sql = "select foo from foo" ;

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);      // get OID
   oid.removeFromSequence("c",1);     // c: {1,3} -> {3}
}
```

removeFromSet

Description

This function is used to remove the corresponding value specified in *value* from the attribute associated with the **SET** constraint on the **CUBRIDOID** instance. If the corresponding value is more than one, the very value found for the first time becomes removed.

Syntax

```
void removeFromSet(String attrName, Object value)
```

Example

```
//create class foo(a set of int, b multiset of int )
//insert into foo values({1,11},{2,13})

String sql = "select foo  from foo"
Connection con = DriverManager.getConnection(url,user,passwd)

Statement stmt = con.createStatement()
CUBRIDResultSet rs= (CUBRIDResultSet) stmt.executeQuery(sql)

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1)                // get OID
    oid.removeFromSet("a",new Integer(11)) // a: {1,11} -> {1}
    oid.removeFromSet("a",new Integer(13)) // b: {2,13} -> {2}
}
```

setReadLock

Description

This function is used to set a read-lock on the instance corresponding to the **CUBRIDOID**.

Syntax

```
void setReadLock()
```

Example

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   oid.setReadLock();
}
```

setValues

Description

This function is used to replace the value specified in the *attrNames* with the value specified in the *values*.

Syntax

```
void setValues(String[] attrNames, Object[] values)
```

Example

```
// create class foo ( a string, b int )
String sql = "select foo from foo";
String[] attr = { "a", "b" };  // a list of attribute names
String[] values = {"CUBRID", new Integer(2001)};

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   oid.setValues(attr, values );
}
```

setWriteLock

Description

This function is used to set a write-lock on the instance corresponding to the **CUBRIDOID**.

Syntax

```
void setWriteLock()
```

Example

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   oid.setWriteLock();
}
```

CUBRIDPreparedStatement

Overview

The **CUBRIDPreparedStatement** class extends the standard **PreparedStatement** and contains the following additional methods.

Return Type	Method Name
CUBRIDOID	executeInsert()
void	setCollection(int index, Object[] array)
void	setOID(int index, CUBRIDOID oid)

executeInsert

Description

This function is used to execute an **INSERT** statement within the **CUBRIDPreparedStatement** object and returns the **CUBRIDOID** corresponding to the inserted object.

Syntax

```
CUBRIDOID executeInsert()
```

Return Value

- A **CUBRIDOID** that corresponds to the inserted object

Example

```
String sql  = "insert into testtable(a) values(?)";

CUBRIDPreparedStatement pstmt = (CUBRIDPreparedStatement)
                                con.prepareStatement(sql);
pstmt.setString(1, "CUBRID");
CUBRIDOID oid = pstmt.executeInsert();
```

setCollection

Description

This function is used to specify the *index*-th parameter in the prepared statement as a collection corresponding to *array*. CUBRID has three types of collections: Set, Multiset and Sequence.

Syntax

```
void setCollection(int index, Object[] array)
```

Example

```
String[] strs = { "abc", "def"};

psmt.setCollection(1, strs);
```

setOID

Description

This function is used to specify the *index*-th parameter in the prepared statement as the **CUBRIDOID** specified in *oid*.

Syntax

```
void setOID(int index, CUBRIDOID oid)
```

CUBRIDResultSet

Overview

The **CUBRIDResultSet** class is extended from the standard **ResultSet** class and has the following additional methods.

Return Type	Method Name
Object	getCollection(int attrIndex)
Object	getCollection(String attrName)
CUBRIDOID	getOid()
CUBRIDOID	getOid(int attrIndex)
CUBRIDOID	getOid(String attrName)

getCollection

Description

This function is used to return the index specified in *attrIndex* or the attribute value specified in *attrName*. The returned object can be converted to an array such as String[].

Syntax

```
Object getCollection(int attrIndex)
Object getCollection(String attrName)
```

Return Value

- An index specified by *attrIndex* or a value of the column that corresponds to the column name specified by *attrName*

getOID

Description

This function is used to return the index specified in *attrIndex* or the attribute value specified in *attrName* to **CUBRIDOID**, thus it returns the **CUBRIDOID**.

If *attrIndex* or *attrName* is not specified, **CUBRIDOID** of the current row of **ResultSet** is returned. This is valid only when **ResultSet** is **TYPE_SCROLL_SENSITIVE** or **CONCUR_UPDATABLE**.

Syntax

```
CUBRIDOID getOID(int attrIndex)
CUBRIDOID getOID(String attrName)
```

```
CUBRIDOID getOID()
```

Return Value

- **CUBRIDOID**

CUBRIDResultSetMetaData

Overview

The **CUBRIDResultSetMetaData** class is extended from the standard **ResultSetMetaData** and has the following additional methods.

Return Type	Method Name
int	getElementType(int columnIndex)
String	getElementTypeName(int columnIndex)

getElementType

Description

This function is used to return a type of the COLLECTION element as *int* defined in the **java.sql.Types**. If a domain of the *columnIndex*-th attribute is not COLLECTION such as **SET**, **MULTISET**, or **SEQUENCE**, **SQLException** occurs in the end.

Syntax

```
int getElementType(int columnIndex)
```

Return Value

- Collection element type (int)

getElementTypeName

Description

This function is used to return the name of the type in the COLLECTION elements. If a domain of the *columnIndex*-th attribute is not COLLECTION such as **SET**, **MULTISET**, or **SEQUENCE**, **SQLException** occurs in the end.

Syntax

```
String getElementTypeName(int columnIndex)
```

Return Value

- Collection element's type name

Example

```
// The following schema is used in this example.
//
// create class foo(
//    a  set(int),
//    b  multiset(int),
//    c  sequence(int)
// );

String sql = "select * from foo" ;
Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
```

```
CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);
CUBRIDResultSetMetaData rsmd = (CUBRIDResultSetMetaData)
                               rs.getMetaData();

int numberofColumn = rsmd.getColumnCount();
for (int i=1; i <= numberofColumn; i++ ) {
   System.out.println(rsmd.getElementType(i) );
   System.out.println(rsmd.getElementTypeName(i) );
}
```

CUBRIDStatement

Overview

The **CUBRIDStatement** class is extended from the standard **Statement** class and has the following additional methods.

Return Type	Method Name
CUBRIDOID	executeInsert(String insertStmt)

executeInsert

Description

This function is used to return the **CUBRIDOID** corresponding to a new tuple (row) inserted by the SQL statement, *insertStmt*.

Syntax

```
CUBRIDOID executeInsert(String insertStmt)
```

Return Value

- **CUBRIDOID** of the added row

Example

```
String sql = "insert into testable(a) values (1)"

CUBRIDStatement stmt = (CUBRIDStatement) con.createStatement();
CUBRIDOID oid = stmt.executeInsert(sql);
```

ODBC API

ODBC Programming

CUBRID ODBC Driver

Description

The CUBRID ODBC driver supports ODBC version 3.52, ODBC core, and some of Level 1 and Level 2 APIs. Since it has been developed based on ODBC Spec 3.x, backward compatibility is not completely ensured for programs written using ODBC Spec 2.x. Only 32 bit are supported. In the 64-bit Windows environment, you can check the ODBC driver with CUBRID 32 bit by executing "C:\Windows\SysWOW64\odbcad32.exe."

For more information on configuring CUBRID ODBC driver, see Configuring the Environment of ODBC and ASP.

Data Type Mapping of CUBRID and ODBC

The following table shows the data mapping relationship between data types of ODBC and those supported by CUBRID.

CUBRID Data Type	ODBC Data Type
Char	SQL_CHAR
Varchar	SQL_VARCHAR
String	SQL_LONGVARCHAR
Nchar	SQL_CHAR
Varnchar	SQL_VARCHAR
Bit	SQL_BINARY
varying bit	SQL_VARBINARY
Numeric	SQL_NUMERIC
Int	SQL_INTEGER
Short	SQL_SMALLINT
Float	SQL_FLOAT
Double	SQL_DOUBLE
Bigint	SQL_BIGINT
Date	SQL_TYPE_DATE
Time	SQL_TYPE_TIME
Timestamp	SQL_TYPE_TIMESTAMP
Datetime	SQL_TYPE_TIMESTAMP
Monetary	SQL_DOUBLE
Oid	SQL_CHAR(32)
set, multiset, sequence	SQL_VARCHAR(MAX_STRING_LENGTH)

Configuring Connection Strings

When you are programming CUBRID ODBC, you can write connection strings as follows:

Item	Example	Description
Driver	CUBRID Driver	Driver name

UID	user1	User ID
PWD	xxx	Password
FETCH_SIZE	100	Fetch size
PORT	30000	Broker port number
SERVER	192.168.1.11	IP address or host name of a CUBRID Broker server
DB_NAME	demodb	Database name
DESCRIPTION	cubrid_test	Description
CHARSET	utf-8	Character set

The following example shows how to use connecting strings above.

```
"DRIVER=CUBRID
Driver;UID=user1;PWD=xxx;FETCH_SIZE=100;PORT=30000;SERVER=192.168.1.11;DB_NAME=demodb;DESC
RIPTION=cubrid_test;CHARSET=utf-8"
```

Supported Functions and Backward Compatibility

Information on supported functions by CUBRID ODBC, versions, compatibility with ODBC Spec is as follows:

API	Version Introduced	Standards Compliance	Support
SQLAllocHandle	3.0	ISO 92	YES
SQLBindCol	1.0	ISO 92	YES
SQLBindParameter	2.0	ODBC	YES
SQLBrowseConnect	1.0	ODBC	NO
SQLBulkOperations	3.0	ODBC	YES
SQLCancel	1.0	ISO 92	YES
SQLCloseCursor	3.0	ISO 92	YES
SQLColAttribute	3.0	ISO 92	YES
SQLColumnPrivileges	1.0	ODBC	NO
SQLColumns	1.0	X/Open	YES
SQLConnect	1.0	ISO 92	YES
SQLCopyDesc	3.0	ISO 92	YES
SQLDescribeCol	1.0	ISO 92	YES
SQLDescribeParam	1.0	ISO 92	NO
SQLDisconnect	1.0	ISO 92	YES
SQLDriverConnect	1.0	ISO 92	YES
SQLEndTran	3.0	ISO 92	YES
SQLExecDirect	1.0	ISO 92	YES
SQLExecute	1.0	ISO 92	YES
SQLFetch	1.0	ISO 92	YES
SQLFetchScroll	3.0	ISO 92	YES
SQLForeignKeys	1.0	ODBC	YES (2008 R3.1 or later)
SQLFreeHandle	3.0	ISO 92	YES
SQLFreeStmt	1.0	ISO 92	YES

SQLGetConnectAttr	3.0	ISO 92	YES
SQLGetCursorName	1.0	ISO 92	YES
SQLGetData	1.0	ISO 92	YES
SQLGetDescField	3.0	ISO 92	YES
SQLGetDescRec	3.0	ISO 92	YES
SQLGetDiagField	3.0	ISO 92	YES
SQLGetDiagRec	3.0	ISO 92	YES
SQLGetEnvAttr	3.0	ISO 92	YES
SQLGetFunctions	1.0	ISO 92	YES
SQLGetInfo	1.0	ISO 92	YES
SQLGetStmtAttr	3.0	ISO 92	YES
SQLGetTypeInfo	1.0	ISO 92	YES
SQLMoreResults	1.0	ODBC	YES
SQLNativeSql	1.0	ODBC	YES
SQLNumParams	1.0	ISO 92	YES
SQLNumResultCols	1.0	ISO 92	YES
SQLParamData	1.0	ISO 92	YES
SQLPrepare	1.0	ISO 92	YES
SQLPrimaryKeys	1.0	ODBC	YES (2008 R3.1 or later)
SQLProcedureColumns	1.0	ODBC	YES (2008 R3.1 or later)
SQLProcedures	1.0	ODBC	YES (2008 R3.1 or later)
SQLPutData	1.0	ISO 92	YES
SQLRowCount	1.0	ISO 92	YES
SQLSetConnectAttr	3.0	ISO 92	YES
SQLSetCursorName	1.0	ISO 92	YES
SQLSetDescField	3.0	ISO 92	YES
SQLSetDescRec	3.0	ISO 92	YES
SQLSetEnvAttr	3.0	ISO 92	NO
SQLSetPos	1.0	ODBC	YES
SQLSetStmtAttr	3.0	ISO 92	YES
SQLSpecialColumns	1.0	X/Open	YES
SQLStatistics	1.0	ISO 92	YES
SQLTablePrivileges	1.0	ODBC	YES (2008 R3.1 or later)
SQLTables	1.0	X/Open	YES

Some functions for which backward compatibility is not supported must be converted into appropriate ones by using the mapping table below.

ODBC 2.x Function	ODBC 3.x Function

SQLAllocConnect	SQLAllocHandle
SQLAllocEnv	SQLAllocHandle
SQLAllocStmt	SQLAllocHandle
SQLBindParam	SQLBindParameter
SQLColAttributes	SQLColAttribute
SQLError	SQLGetDiagRec
SQLFreeConnect	SQLFreeHandle
SQLFreeEnv	SQLFreeHandle
SQLFreeStmt with SQL_DROP	SQLFreeHandle
SQLGetConnectOption	SQLGetConnectAttr
SQLGetStmtOption	SQLGetStmtAttr
SQLParamOptions	SQLSetStmtAttr
SQLSetConnectOption	SQLSetConnectAttr
SQLSetParam	SQLBindParameter
SQLSetScrollOption	SQLSetStmtAttr
SQLSetStmtOption	SQLSetStmtAttr
SQLTransact	SQLEndTran

Using OIDs and Collections

ODBC is designed for relational DBMSs. Therefore, CUBRID ODBC does not support some object-oriented features such as CUBRID OIDs and collections. It is because CUBRID is an object-relational DBMS that integrates relational and object-oriented data models.

Using OIDs

Because the CUBRID ODBC driver considers an OID as a string (char(32)), the **INSERT**, **UPDATE** and **DELETE** statements containing OIDs can be used as follows. The OID string should be used with single quotes (''). The domain of the member attribute in the following example is the same as the OID.

```
insert into foo(member) values('@12|34|56')
delete from foo where member = '@12|34|56'
update foo set age = age + 1 where member = '@12|34|56'
```

Using Collections

Collection types : **SET**, **MULTISET** and **SEQUENCE** are supported. The CUBRID ODBC driver considers a collection as a string (longvarchar). You can obtain a collection by separating each element in the **SELECT** statement using commas in braces as with "{value_1, value_2, ...value_n}."

Note If a string longer than defined size in a column is inserted(INSERT) or updated(UPDATE), the string will be truncated.

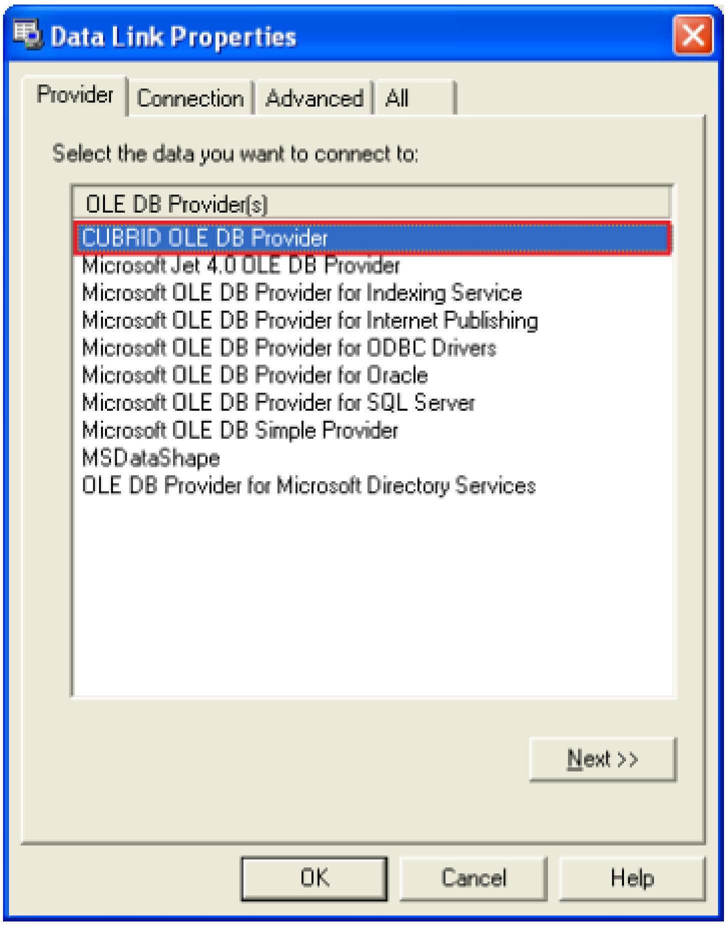
OLE DB API

OLE DB Programming

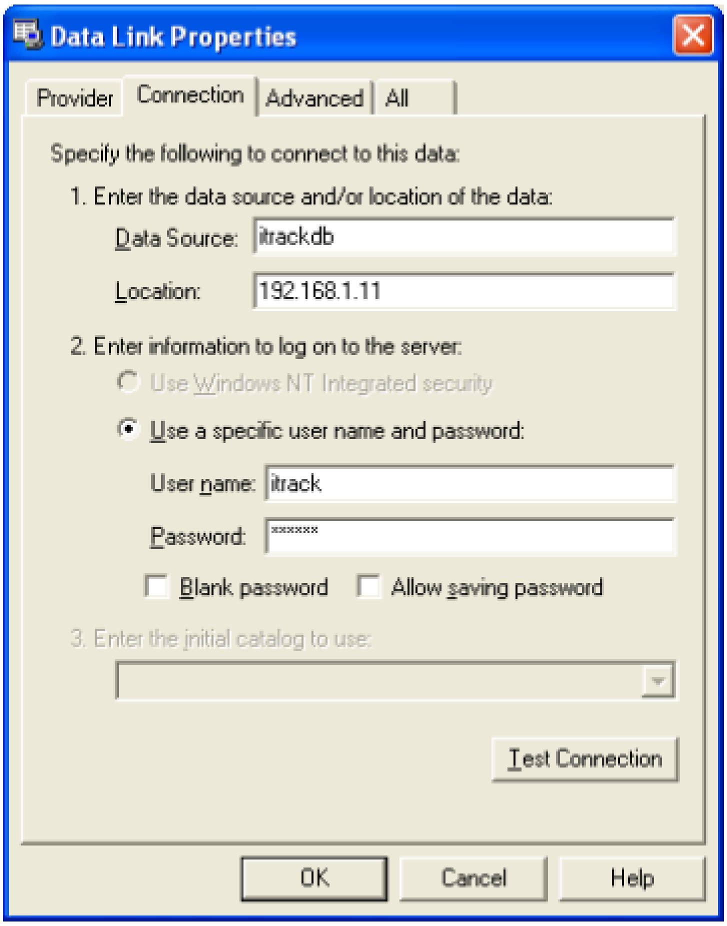
Using Data Link Property Dialog Box

In the [Data Link Properties] dialog box, you can check and configure various OLE DB providers provided by the current Windows operating system.

If you have properly installed the CUBRID OLE DB Provider for Windows, 'CUBRID OLE DB Provider' is displayed in the provider list of the [Data Link Properties] dialog box, as shown below.

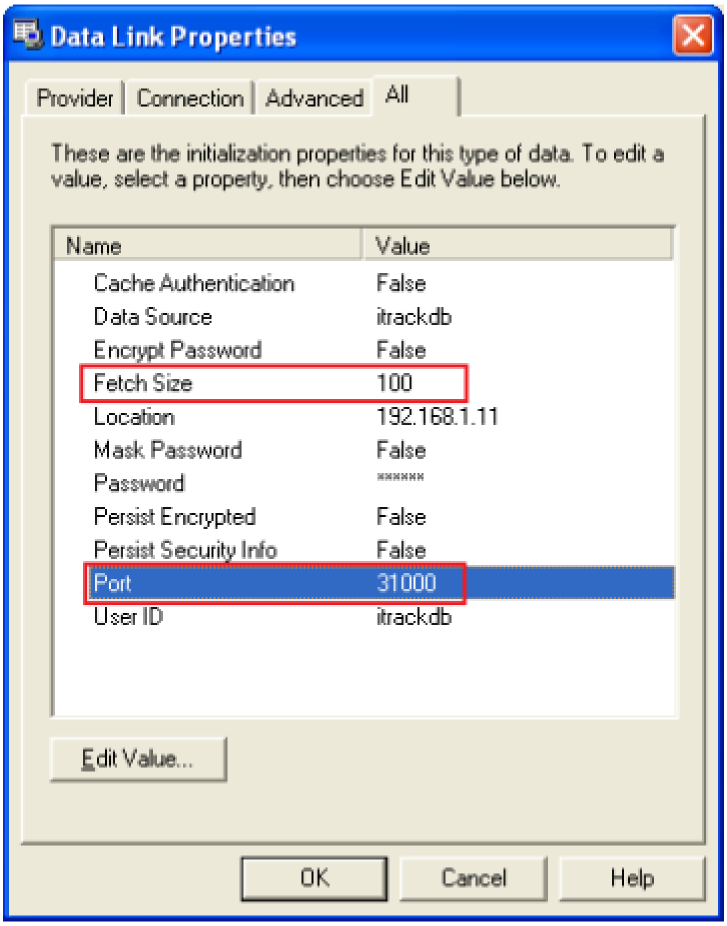


If you click the [Next] button after selecting 'CUBRID OLE DB Provider', the [Connection] tab appears as shown below. Set the desired link properties in the [Connection] tab.



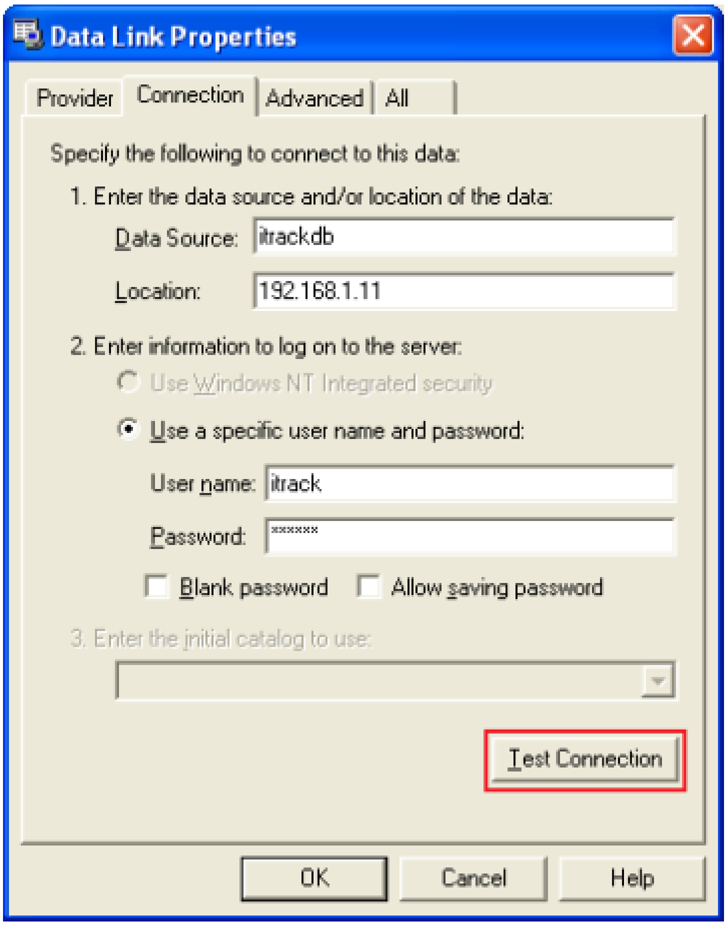
- **Data source** : Enter the name of the CUBRID database.
- **Location** : Enter the IP address or host name of the server where the CUBRID Broker is running.
- **User name** : Enter the name of the user who will log on to the database server.
- **Password** : Enter the password to be used for the database server logon.

Select all connection properties and then click the [All] tab.



To check every value currently configured, click the [All] tab; to edit the value, double-click the item you want. When the [Edit Property Value] dialog box appears, enter the desired value and then click [OK]. The figure above shows an example that configures the [Port] to "31000," and [Fetch Size] to "100."

You can check whether the connection is working properly by clicking the [Test Connection] button in the [Connection] tab after completing all configuration.



Configuring Connection String

When you program the CUBRID OLE DB Provider using ADO (ActiveX Data Object) or ADO.net, write the connection string as follows:

Item	Example	Description
Provider	CUBRIDProvider	Provider name
Data source	demodb	Database name
Location	192.168.1.11	The IP address of the CUBRID Broker Server
User ID	PUBLIC	User ID
Password	xxx	Password
Port	30000	Broker port number
Fetch Size	100	Fetch size

A connection string using the above example is as follows:

```
"Provider = CUBRIDProvider;Data Source = demodb;Location = 192.168.1.11;User ID =
PUBLIC;Password =xxx;Port = 30000;Fetch Size = 100"
```

Multi-Thread Programming in .NET Environment

To develop programs by using the CUBRID OLE DB Provider in the Microsoft .NET, you should consider the followings:

If you develop multi-thread programs by using ADO.NET in the management environment, you need to change the value of the ApartmentState attribute of the Thread object to a ApartmentState.STA value because the CUBRID OLE DB Provider supports only Single Threaded Apartment (STA) attributes.

Without any change of given values, the default value of the attribute in the Thread object returns Unknown value, thereby causing abnormal process or errors during multi-threads programming.

Caution All OLE DB objects are COM objects. Currently, the CUBRID OLE DB Provider supports only the apartment threading model among COM threading models. It does not support the free threading model. This applies to not only the .NET but all multi-threaded environment.

Note If a string longer than defined size in a column is inserted(INSERT) or updated(UPDATE), the string will be truncated.

PHP API

PHP Programming

General Features

Connection

- Connecting to a database : The first step of a database application is to use the cubrid_connect() or cubrid_connect_with_url() function which provide a database connection. Once the cubrid_connect() or cubrid_connect_with_url() function is executed successfully, you can use any functions available in the database. It is very important to call the cubrid_disconnect() function before the application is terminated completely. The cubrid_disconnect() function terminates the current transaction as well as the connection handle and all request handles created by the cubrid_connect() function.

Transactions and auto-commit

CUBRID PHP supports both transaction and auto-commit mode. Auto-commit mode means that every query that you run has its own implicit transaction. You can use the cubrid_get_autocommit() function to get the status of current connection auto-commit mode, and use the cubrid_set_autocommit() function to enable/disable auto-commit mode of current connection. When cubrid_set_autocommit() function is called, concurrent transactions are committed regardless of the auto-commit mode. The default mode of autocommit-mode is off. You can also use cubrid_connect_with_url() function to set the autocommit-mode when you establish the database connection. For example:

```
$con = cubrid_connect_with_url("cci:CUBRID:localhost:33000:demodb:dba::?autocommit=true");
```

If you need a transaction, you must use the cubrid_set_autocommit() function to disable the auto-commit mode. The cubrid_commit() or cubrid_rollback() function is used to commit or roll back a transaction. The cubrid_disconnect() function terminates the transaction and rolls back uncommitted ones.

Processing Queries

The following are basic steps of query execution.

- Creating a connection handle
- Creating a request handle for an SQL query request
- Fetching the result
- Terminating the request handle

```
$con = cubrid_connect("192.168.0.10", 33000, "demodb");
if($con) {
      $req = cubrid_execute($con, "select * from code")
      if($req) {
              while ($row = cubrid_fetch($req)) {
              echo $row["s_name"];
              echo $row["f_name"];
              }
              cubrid_close_request($req);
      }
      cubrid_disconnect($con);
}
```

Column types and names of the query result

The cubrid_column_types() function is used to get an array containing column types, and the cubrid_column_names() function is used to get an array containing column names.

```
$req = cubrid_execute($con, "select host_year, host_city from olympic");
if($req) {
      $coltypes = cubrid_column_types($req);
      $colnames = cubrid_column_names($req);

      while (list($key, $coltype) = each ($col_types)) {
```

```
      echo $col_type;
     }
      while (list($key, $colname) = each ($col_names))
      echo $col_name;

      cubrid_close_request($req);
}
```

Adjusting the cursor

You can configure the position of the query result. The cubrid_move_cursor() function is used to move the cursor to a certain position from one of three points: the beginning of the query result, the current cursor position and the end of the query result.

```
$req = cubrid_execute($con, "select host_year, host_city from olympic order by host_year");
if($req) {
cubrid_move_cursor($req, 20, CUBRID_CURSOR_CURRENT)
    while ($row = cubrid_fetch($req, CUBRID_ASSOC)) {
    echo $row["host_year"]." ";
    echo $row["host_city"]."\n"; }
}
```

Result array types

One of the following three types of arrays is used in the result of the cubrid_fetch() function. The type of the array can be determined when the cubrid_fetch() function is called. The associative array uses character string indexes. The numeric array uses numeric order indexes. The last array type includes both associative and numeric arrays.

- Numeric array

```
while (list($host_year, $host_city) = cubrid_fetch($req, CUBRID_NUM)) {
      echo $host_year;
    echo $host_city;
}
```

- Associative array

```
while ($row = cubrid_fetch($req, CUBRID_ASSOC)) {
      echo $row["host_year"];
      echo $row["host_city"];
}
```

Catalog Operation

Information about the database schema such as classes, virtual classes, attributes, functions, triggers and constraints can be obtained by calling the cubrid_schema() function. The return value of the cubrid_schema() function is a two-dimensional array.

```
$pk =cubrid_schema($con, CUBRID_SCH_PRIMARY_KEY,"game");
if ($pk) {
print_r($pk);
}

$fk = cubrid_schema($con, CUBRID_SCH_IMPORTED_KEYS, "game");
if ($fk) {
print_r($fk);
}
```

Processing Errors

When an error occurs, most PHP interface functions display the error message and return false or -1. Each error message, error code or error facility code can be checked by using the cubrid_error_msg(), cubrid_error_code(), and cubrid_error_code_facility() functions.

The return value of the cubrid_error_code_facility() function is one of **CUBRID_FACILITY_DBMS** (DBMS error), **CUBRID_FACILITY_CAS** (CAS server error), **CUBRID_FACILITY_CCI** (CCI error) and **CUBRID_FACILITY_CLIENT** (PHP module error).

CUBRID Features

Using OIDs

With a query that can update the **CUBRID_INCLUDE_OID** option in the cubrid_execute() function, you can get the OID value of the current row updated by the executing cubrid_current_oid().

```
$req =cubrid_execute($con,"select * from person where id =1", CUBRID_INCLUDE_OID);
if ($req) {
while ($row = cubrid_fetch($req)) {
echo cubrid_current_oid($req);
echo $row["id"];
echo $row["name"];
}

cubrid_close_request($req);
}
```

You can get all attributes, the specified attribute or an attribute of an instance by using the OID.

If you don't specify any attribute in the cubrid_get() function, the values of all attributes are returned (a). If you specify an attribute as an array data type, an associative array containing the values of the specified attribute is returned (b). If you specify an attribute as a character string array, the value of the attribute is returned (c).

```
$attr_array = cubrid_get($con, $oid); // (a)
$attr_array = cubrid_get($con, $oid, array("id", "name")); // (b)
$attr_array = cubrid_get($con, $oid, "id"); // (c)
```

You can also update an attribute value of an instance by using the OID. To update a single attribute value, specify the attribute name as a character string type and its value (a). To set multiple attribute values, specify an associative array containing the attribute names and values (b).

```
$cubrid_put ($con, $oid, "id", 1); // (a)
$cubrid_put ($con, $oid, array("id"=>1, "name"=>"Tomas")); // (b)
```

Using Collections

- Collection data types can be used by using either PHP array data types or PHP functions that support array data types. The following is an example of fetching the query result with the cubrid_fetch() function.

```
$row = cubrid_fetch ($req);
$col = $row["customer"];
while (list ($key, $cust) = each ($col)) {
echo $cust;
}
```

- You can also get values of collection attributes. The following is an example of getting collection attribute values with the cubrid_col_get() function.

```
$tels = cubrid_col_get ($con, $oid, "tels");
while (list ($key, $tel) = each ($tels)) {
echo $tel."\n";
}
```

- You can directly update collection type values with cubrid_set_add() and cubrid_set_drop() functions.

```
$tels = cubrid_col_get ($con, $oid, "tels");
while (list ($key, $tel) = each ($tels)) {
$res = cubrid_set_drop ($con, $oid, "tel", $tel);
}
cubrid_commit ($con);
```

cubrid_affected_rows

Description

The **cubrid_affected_rows**() function is used to get the number of rows that have been affected by the SQL statements (**INSERT**, **DELETE**, and **UPDATE**).

Syntax

```
int cubrid_affected_rows([ resource $req_identifier ])
```

- *req_identifier* : Request identifier. If the request identifier is not specified, the last request is assumed.

Return Value

- Success : Returns the number of rows affected by the SQL statement.
- When last SQL statement is not INSERT, UPDATE or DELETE : -1
- When request identifier is not specified and there is no last request : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE cubrid_test");
cubrid_execute($conn, "CREATE TABLE cubrid_test (t varchar)");

for ($i = 0; $i < 5; $i++) {
    cubrid_execute($conn, "INSERT INTO cubrid_test(t) VALUES('cubrid_test')");
}

cubrid_execute($conn, "DELETE FROM cubrid_test");

$affected_num = cubrid_affected_rows();
var_dump($affected_num);

cubrid_disconnect($conn);
?>

The above example will output:

int(5)
```

See Also

- cubrid_execute

cubrid_bind

Description

The **cubrid_bind**() function is used to substitute a value for a variable of the cubrid_prepare() with parameters, a various types in PHP and corresponding types in SQL. If bind_value_type is not given, string will be the default. The following table shows the types of substitute values:

Support	Bind type	Corresponding SQL type
Supported	STRING	CHAR, VARCHAR
	NCHAR	NCHAR, NVARCHAR
	BIT	BIT, VARBIT
	NUMERIC or NUMBER	SHORT, INT, NUMERIC
	FLOAT	FLOAT
	DOUBLE	DOUBLE
	TIME	TIME
	DATE	DATE
	TIMESTAMP	TIMESTAMP
	OBJECT	OBJECT

	BLOB	BLOB
	CLOB	CLOB
	NULL	NULL
Not supported	SET	SET
	MULTISET	MULTISET
	SEQUENCE	SEQUENCE

Syntax

```
bool cubrid_bind(resource $req_identifier,mixed $bind_param, mixed $bind_value[,string
$bind_value_type])
```

- *req_identifier* : Request identifier as a result of cubrid_prepare()
- *bind_param* : Parameter identifier. For a prepared statement using named placeholders, this will be a parameter name of the form :name (Note that the name can only contain digit, alphabet, and underscore, and it cannot begin with digit. The name length cannot be longer than 32). For a prepared statement using question mark placeholders, this will be the 1-indexed position of the parameter.
- *bind_ value* : Actual value to be bound
- *bind_value_type* : Type of the value to be bound. It can be omitted by default. If it is omitted, the type is automatically cast to an appropriate one. However, **NCHAR**, **BLOB**/**CLOB** and **BIT** types must be passed as arguments.

Note If data to be bound is **BLOB**/**CLOB**, CUBRID will map the data as a PHP stream, which introduces a unified approach to the handling of files and sockets in PHP extension. If the actually bind value type is not stream, CUBRID will convert it to string.

Return Value

- Success : TRUE
- Failure : FALSE

Example 1

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$result = cubrid_execute($conn, "SELECT code FROM event WHERE sports='Basketball' and
gender='M'");
$row = cubrid_fetch_array($result, CUBRID_ASSOC);
$event_code = $row["code"];

cubrid_close_request($result);

$game_req = cubrid_prepare($conn, "SELECT athlete_code FROM game WHERE host_year=1992 and
event_code=? and nation_code='USA'");
cubrid_bind($game_req, 1, $event_code, "number");
cubrid_execute($game_req);

printf("--- Dream Team (1992 United States men's Olympic basketball team) ---\n");
while ($athlete_code = cubrid_fetch_array($game_req, CUBRID_NUM)) {
    $athlete_req = cubrid_prepare($conn, "SELECT name FROM athlete WHERE code=? AND
nation_code='USA' AND event='Basketball' AND gender='M'");
    cubrid_bind($athlete_req, 1, $athlete_code[0], "number");
    cubrid_execute($athlete_req);
    $row = cubrid_fetch_assoc($athlete_req);
    printf("%s\n", $row["name"]);
}
cubrid_close_request($game_req);
cubrid_close_request($athlete_req);
cubrid_disconnect($conn);
?>

The above example will output:
```

```
--- Dream Team (1992 United States men's Olympic basketball team) ---
Stockton John
Robinson David
Pippen Scottie
Mullin C.
Malone Karl
Laettner C.
Jordan Michael
Johnson Earvin
Ewing Patrick
Drexler Clyde
Bird Larry
Barkley Charles
```

Example 2

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$sql_stmt = <<<EOD
SELECT s.name FROM stadium s, game g
WHERE s.code = g.stadium_code AND g.medal = :medal_type
GROUP BY g.stadium_code ORDER BY count(medal) DESC LIMIT 1;
EOD;

$req = cubrid_prepare($conn, $sql_stmt);
printf("%-30s %s\n", "Medal Type", "Stadium where most medals were ever won");

cubrid_bind($req, ":medal_type", "G");
cubrid_execute($req);
$row = cubrid_fetch_assoc($req);
printf("%-30s %s\n", "Gold", $row["name"]);

cubrid_bind($req, ":medal_type", "S");
cubrid_execute($req);
$row = cubrid_fetch_assoc($req);
printf("%-30s %s\n", "Silver", $row["name"]);

cubrid_bind($req, ":medal_type", "B");
cubrid_execute($req);
$row = cubrid_fetch_assoc($req);
printf("%-30s %s\n", "Bronze", $row["name"]);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

Medal Type                     Stadium where most medals were ever won
Gold                           Olympic Aquatic Centre
Silver                         Olympic Aquatic Centre
Bronze                         Sydney Convention and Exhibition Centre
```

Example 3

```
<?php
$con = cubrid_connect("localhost", 33000, "foo");
if ($con) {
    $sql = "INSERT INTO php_cubrid_lob_test(doc_content) VALUES(?)";
    $req = cubrid_prepare($con, $sql);

    $fp = fopen("book.txt", "rb");

    cubrid_bind($req, 1, $fp, "blob");
    cubrid_execute($req);
}
?>
```

Example 4

```
<?php
```

```
$con = cubrid_connect("localhost", 33000, "foo");
if ($con) {
    $sql = "INSERT INTO php_cubrid_lob_test(image) VALUES(?)";
    $req = cubrid_prepare($con, $sql);

    cubrid_bind($req, 1, "cubrid_logo.png", "blob");
    cubrid_execute($req);
}
?>
```

See Also

- cubrid_execute
- cubrid_close_request

cubrid_client_encoding

Description

The **cubrid_client_encoding**() function returns the current CUBRID connection charset.

Syntax

```
string cubrid_client_encoding ([ resource $conn_identifier ])
```

- *conn_identifier* : The CUBRID connection. If the connection identifier is not specified, the last connection opened is assumed.

Return Value

- Success : A string that represents the CUBRID connection charset
- Failure : FALSE

Example

```
<?php
    $con = cubrid_connect("localhost", 33000, "demodb");
    if (!$con)
    {
        die('Could not connect.');
    }

    printf("CUBRID current charset: %s\n", cubrid_client_encoding($con));
?>
```

See Also

- cubrid_get_charset

cubrid_close

Description

The **cubrid_close**() function is used to stop transactions currently being executed, terminate the connection with the server and close the connection handle. All request handles that are still open will be closed.

Syntax

```
bool cubrid_close ([resource $con_identifier])
```

- *con_identifier* : Connection identifier. If the connection identifier is not specified, the last connection opened is assumed.

Return Value

- Success : TRUE

- Failure : FALSE

Example

```
$con = cubrid_connect("192.168.0.10", 33000, "demodb");
if ($con) {
    echo "connected successfully";
    $req = cubrid_execute( $con, "insert into person values(1,'James')");
    if ($req) {
        cubrid_close_request($req);
        cubrid_commit($con);
    } else {
        cubrid_rollback($con);
    }
    cubrid_close($con);
}
```

See Also

- cubrid_connect

cubrid_close_prepare, cubrid_close_request

Description

The **cubrid_close_prepare**() function or the **cubrid_close_request**() function is used to close the request handle given to the *req_identifier* parameter and release the memory area related to the handle.

Syntax

```
int cubrid_close_prepare (resource $req_identifier)
int cubrid_close_request (resource $req_identifier)
```

- *req_identifier* : Request identifier

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = cubrid_prepare ($conn, "SELECT * FROM olympic WHERE host_year=?");

$host_year = 2004;
cubrid_bind($req, 1, $host_year, "number");
cubrid_execute($req);

printf("%-9s %-11s %-9s %-12s %-12s %-15s %-15s\n",
    "host_year", "host_nation", "host_city", "opening_date",
    "closing_date", "mascot", "slogan");

while ($row = cubrid_fetch_assoc($req)) {
    printf("%-9s %-11s %-9s %-12s %-12s %-15s %-15s\n",
        $row["host_year"], $row["host_nation"], $row["host_city"],
        $row["opening_date"], $row["closing_date"], $row["mascot"], $row["slogan"]);
}

cubrid_close_request($req);

cubrid_disconnect($conn);
?>

The above example will output:

host_year host_nation host_city opening_date closing_date mascot         slogan
2004      Greece      Athens    2004-8-13    2004-8-29    Athena  Phevos  Welcome Home
```

See Also

- cubrid_execute

cubrid_close_prepare, cubrid_close_request

Description

The **cubrid_close_prepare**() function or the **cubrid_close_request**() function is used to close the request handle given to the *req_identifier* parameter and release the memory area related to the handle.

Syntax

```
int cubrid_close_prepare (resource $req_identifier)
int cubrid_close_request (resource $req_identifier)
```

- *req_identifier* : Request identifier

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = cubrid_prepare ($conn, "SELECT * FROM olympic WHERE host_year=?");

$host_year = 2004;
cubrid_bind($req, 1, $host_year, "number");
cubrid_execute($req);

printf("%-9s %-11s %-9s %-12s %-12s %-15s %-15s\n",
    "host_year", "host_nation", "host_city", "opening_date",
    "closing_date", "mascot", "slogan");

while ($row = cubrid_fetch_assoc($req)) {
    printf("%-9s %-11s %-9s %-12s %-12s %-15s %-15s\n",
        $row["host_year"], $row["host_nation"], $row["host_city"],
        $row["opening_date"], $row["closing_date"], $row["mascot"], $row["slogan"]);
}

cubrid_close_request($req);

cubrid_disconnect($conn);
?>

The above example will output:

host_year host_nation host_city opening_date closing_date mascot          slogan
2004      Greece      Athens    2004-8-13    2004-8-29    Athena  Phevos  Welcome Home
```

See Also

- cubrid_execute

cubrid_col_get

Description

The **cubrid_col_get**() function is used to get the elements of the given collection type (set, multiset, sequence) attribute in the form of an array.

Syntax

```
array cubrid_col_get (resource $conn_identifier, string $oid, string $attr_name)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the desired instance
- *attr_name* : Name of the attribute to be read from the instance

Return Value

- Success : An array that contains the desired elements (0 : default numeric array)
- Failure : FALSE. If an error occurs, a warning message is displayed to distinguish it from a collection without attributes or **NULL**. You can check the error with cubrid_error_code().

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_col_get($conn, $oid, "b");
var_dump($attr);

$size = cubrid_col_size($conn, $oid, "b");
var_dump($size);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "3"
}
int(3)
```

cubrid_col_size

Description

The **cubrid_col_size**() function is used to get the number of elements of a collection type (set, multiset, sequence) attribute.

Syntax

```
int cubrid_col_size(resource $conn_identifier, string $oid, string $attr_name)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance

Return Value

- Success : The number of elements
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_col_get($conn, $oid, "b");
var_dump($attr);

$size = cubrid_col_size($conn, $oid, "b");
var_dump($size);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "3"
}
int(3)
```

cubrid_column_names

Description

The **cubrid_column_names**() function is used to get column names in the query results by using *req_identifier*.

Syntax

```
array cubrid_column_names (resource $req_identifier)
```

- *req_identifier* : Request identifier

Return Value

- Success : An array that contains the column names
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$result = cubrid_execute($conn, "SELECT * FROM game WHERE host_year=2004 AND
nation_code='AUS' AND medal='G'");

$column_names = cubrid_column_names($result);
$column_types = cubrid_column_types($result);

printf("%-30s %-30s %-15s\n", "Column Names", "Column Types", "Column Maxlen");
for($i = 0, $size = count($column_names); $i < $size; $i++) {
    $column_len = cubrid_field_len($result, $i);
    printf("%-30s %-30s %-15s\n", $column_names[$i], $column_types[$i], $column_len); }
```

```
cubrid_disconnect($conn);
?>

The above example will output:

Column Names                 Column Types                 Column Maxlen
host_year                    integer                      11
event_code                   integer                      11
athlete_code                 integer                      11
stadium_code                 integer                      11
nation_code                  char(3)                      3
medal                        char(1)                      1
game_date                    date                         10
```

See Also

- cubrid_execute
- cubrid_prepare
- cubrid_column_types

cubrid_column_types

Description

The **cubrid_colulmn_types**() function is used to get column types in the query results by using *req_identifier*.

Syntax

```
array cubrid_column_types (resource $req_identifier)
```

- *req_identifier* : Request identifier

Return Value

- Success : An array that contains the column types
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$result = cubrid_execute($conn, "SELECT * FROM game WHERE host_year=2004 AND
nation_code='AUS' AND medal='G'");

$column_names = cubrid_column_names($result);
$column_types = cubrid_column_types($result);

printf("%-30s %-30s %-15s\n", "Column Names", "Column Types", "Column Maxlen");
for($i = 0, $size = count($column_names); $i < $size; $i++) {
    $column_len = cubrid_field_len($result, $i);
    printf("%-30s %-30s %-15s\n", $column_names[$i], $column_types[$i], $column_len); }

cubrid_disconnect($conn);
?>

The above example will output:

Column Names                 Column Types                 Column Maxlen
host_year                    integer                      11
event_code                   integer                      11
athlete_code                 integer                      11
stadium_code                 integer                      11
nation_code                  char(3)                      3
medal                        char(1)                      1
game_date                    date                         10
```

See Also

- cubrid_execute
- cubrid_prepare
- cubrid_column_names

cubrid_commit

Description

The **cubrid_commit**() function is used to commit on the transaction pointed by *conn_identifier*, currently in progress. Connection to the server is closed after the **cubrid_commit**() function is called; the connection handle is still valid, though.

In CUBRID PHP, an auto-commit mode is enabled by default for transaction management. If you want to set auto-commit to off when a new transaction starts, you must use the cubrid_set_autocommit() function.

Syntax

```
bool cubrid_commit (int $conn_identifier)
```

- *conn_identifier* : Connection identifier

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("127.0.0.1", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE publishers");

$sql = <<EOD
CREATE TABLE publishers(
pub_id CHAR(3),
pub_name VARCHAR(20),
city VARCHAR(15),
state CHAR(2),
country VARCHAR(15)
)
EOD;

if (!cubrid_execute($conn, $sql)) {
    printf("Error facility: %d\nError code: %d\nError msg: %s\n",
cubrid_error_code_facility(), cubrid_error_code(), cubrid_error_msg());

    cubrid_disconnect($conn);
    exit;
}

$req = cubrid_prepare($conn, "INSERT INTO publishers VALUES(?, ?, ?, ?, ?)");

$id_list = array("P01", "P02", "P03", "P04");
$name_list = array("Abatis Publishers", "Core Dump Books", "Schadenfreude Press",
"Tenterhooks Press");
$city_list = array("New York", "San Francisco", "Hamburg", "Berkeley");
$state_list = array("NY", "CA", NULL, "CA");
$country_list = array("USA", "USA", "Germany", "USA");

for ($i = 0, $size = count($id_list); $i < $size; $i++) {
    cubrid_bind($req, 1, $id_list[$i]);
    cubrid_bind($req, 2, $name_list[$i]);
    cubrid_bind($req, 3, $city_list[$i]);
    cubrid_bind($req, 4, $state_list[$i]);
    cubrid_bind($req, 5, $country_list[$i]);
```

```
    if (!($ret = cubrid_execute($req))) {
        break;
    }
}

if (!$ret) {
    cubrid_rollback($conn);
} else {
    cubrid_commit($conn);

    $req = cubrid_execute($conn, "SELECT * FROM publishers");
    while ($result = cubrid_fetch_assoc($req)) {
        printf("%-3s %-20s %-15s %-3s %-15s\n",
            $result["pub_id"], $result["pub_name"], $result["city"], $result["state"],
$result["country"]);
    }
}

cubrid_disconnect($conn);
?>

The above example will output:

P01     Abatis Publishers     New York         NY     USA
P02     Core Dump Books       San Francisco    CA     USA
P03     Schadenfreude Press   Hamburg                 Germany
P04     Tenterhooks Press     Berkeley         CA     USA
```

See Also

- cubrid_rollback

cubrid_connect

Description

The **cubrid_connect**() function is used to configure the connection environment with the server by using the given information such as the server address, port number, database name, user name and password. If the user name and password are not set, **PUBLIC** is used as default.

Syntax

```
resource cubrid_connect(string $host, int $port, string $dbname[, string $userid[, string
$passwd]])
```

- *host* : IP address and host name of the Broker Server
- *port* : Port number of the Broker Server
- *dbname* : Database name
- *userid* : Database user name
- *passwd* : Database user password

Return Value

- Success : Connection handle
- Failure : FALSE

Example

```
<?php
printf("%-30s %s\n", "CUBRID PHP Version:", cubrid_version());

printf("\n");
$conn = cubrid_connect("localhost", 33000, "demodb");

if (!$conn) {
    die('Connect Error ('. cubrid_error_code() .')' . cubrid_error_msg());
```

```
}

$db_params = cubrid_get_db_parameter($conn);

while (list($param_name, $param_value) = each($db_params)) {
    printf("%-30s %s\n", $param_name, $param_value);
}

printf("\n");

$server_info = cubrid_get_server_info($conn);
$client_info = cubrid_get_client_info();

printf("%-30s %s\n", "Server Info:", $server_info);
printf("%-30s %s\n", "Client Info:", $client_info);

printf("\n");

$charset = cubrid_get_charset($conn);

printf("%-30s %s\n", "CUBRID Charset:", $charset);

cubrid_disconnect($conn);
?>
The above example will output:

CUBRID PHP Version:          8.3.1.0005

PARAM_ISOLATION_LEVEL       3
LOCK_TIMEOUT                -1
MAX_STRING_LENGTH           1073741823
PARAM_AUTO_COMMIT           0
Server Info:                8.3.1.0173
Client Info:                8.3.1

CUBRID Charset:             iso8859-1
```

See Also

- cubrid_disconnect

cubrid_connect_with_url

Description

The **cubrid_connect_with_url**() function tries to connect a database by using connection information passed with an url string argument. If the HA feature is enabled in PHP, you must specify connection information of the active server and connection information of the standby server, which is used for failover when failure occurs, in the *url* string argument of this function. If it has succeeded, the ID of connection handle is returned; if it fails, an error code is returned.

Syntax

```
resource cubrid_connect_with_url(string $conn_url[, string $db_user, string $db_password ])
```

- *conn_url* : A character string that contains server connection information
- *db_user* : A name of the database user
- *db_password* : A database user password

```
<conn_url> ::=cci:cubrid:<host>:<db_name>:<db_user>:<db_password>:[?<properties>]
<properties> ::= <property> [&<property>]
<property> ::= autocommit=<autocommit_mode>
<property> ::= althosts=<alternative_hosts> [&rctime=<time>]
<alternative_hosts> ::= <standby_broker1_host>:<port> [,<standby_broker2_host>:<port>]

<host> := HOSTNAME | IP_ADDR
<time> := SECOND
```

- *host* : A host name or IP address of the master database

- *db_name* : A name of the database
- **autocommit**=*true*/*false* : The database connection auto commit mode.
- **althosts** =*standby_broker1_host, standby_broker2_host, . . .* : Specifies the broker information of the standby server, which is used for failover when it is impossible to connect to the active server. You can specify multiple brokers for failover, and the connection to the brokers is attempted in the order listed in **alhosts**.
- **rctime** : An interval between the attempts to connect to the active broker in which failure occurred. After a failure occurs, the system connects to the broker specified by **althosts** (failover), terminates the transaction, and then attempts to connect to the active broker of the master database at every **rctime**. The default value is 600 seconds.
- *db_user* : (IN) A name of the database user
- *db_passwd* : (IN) A database user password

Return Value

- Success : Connection identifier
- Failure : FALSE

Example

```
<?php
$con = cubrid_connect_with_url("cci:CUBRID:localhost:33000:demodb:dba::?autocommit=true");
?>
```

cubrid_current_oid

Description

The **cubrid_current_oid**() function is used to get the OID of the current cursor location from the query result. To use **cubrid_current_oid**(), the query executed must be an updatable query, and the **CUBRID_INCLUDE_OID** option must be included during the query execution.

Syntax

```
string cubrid_current_oid (resource $req_identifier)
```

- *req_identifier* : Request identifier

Return Value

- Success : OID of the current cursor position
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$req = cubrid_execute($conn, "SELECT * FROM code", CUBRID_INCLUDE_OID);
$oid = cubrid_current_oid($req);
$res = cubrid_get($conn, $oid);

print_r($res);

cubrid_disconnect($conn);
?>

The above example will output:

Array
(
    [s_name] => X
    [f_name] => Mixed
)
```

See Also

- cubrid_execute

cubrid_data_seek

Description

The **cubrid_data_seek**() function moves the internal row pointer of the CUBRID result associated with the specified result identifier to point to the specified *row_number*. The next call to a CUBRID fetch function, such as cubrid_fetch_assoc(), would return that row.

Syntax

```
bool cubrid_data_seek (resource $req_identifier, int $row_number)
```

- *req_identifier* : Result identifier
- *row_number* : The desired row number of the new result pointer

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("127.0.0.1", 33000, "demodb");

$req = cubrid_execute($conn, "SELECT * FROM code");
cubrid_data_seek($req, 0);

$result = cubrid_fetch_row($req);
var_dump($result);

cubrid_data_seek($req, 2);
$result = cubrid_fetch_row($req);
var_dump($result);

cubrid_data_seek($req, 4);
$result = cubrid_fetch_row($req);
var_dump($result);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(2) {
    [0]=>
    string(1) "X"
    [1]=>
    string(5) "Mixed"
}
array(2) {
    [0]=>
    string(1) "M"
    [1]=>
    string(3) "Man"
}
array(2) {
    [0]=>
    string(1) "S"
    [1]=>
    string(6) "Silver"
}
```

cubrid_ db_name

Description

The **cubrid_db_name**() functions is used to get db name from results of cubrid_list_dbs().

Syntax

```
string cubrid_db_name(resource $result, int $index)
```

- *result* : The result from a call to cubrid_list_dbs
- *index* : The index into the result set

Return Value

- Success : database name
- Failure : The index into the result set

Example

```
<?php
error_reporting(E_ALL); $conn = cubrid_connect('dbhost', 33000, 'demodb');
$db_list = cubrid_list_dbs($conn);

$i = 0;
$cnt = cubrid_num_rows($db_list);
while ($i < $cnt) {
echo cubrid_db_name($db_list, $i) . "\n";
    $i++;
}
?>
```

See Also

- cubrid_list_dbs

cubrid_disconnect

Description

The **cubrid_disconnect**() function is used to stop transactions currently being executed, terminate the connection with the server and close the connection handle. All request handles that are still open will be closed.

Syntax

```
bool cubrid_disconnect (resource $conn_identifier)
```

- *conn_identifier* : Connection identifier

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
printf("%-30s %s\n", "CUBRID PHP Version:", cubrid_version());

printf("\n");

$conn = cubrid_connect("localhost", 33000, "demodb");

if (!$conn) {
    die('Connect Error ('. cubrid_error_code() .')' . cubrid_error_msg());
}
```

```
$db_params = cubrid_get_db_parameter($conn);

while (list($param_name, $param_value) = each($db_params)) {
    printf("%-30s %s\n", $param_name, $param_value);
}

printf("\n");

$server_info = cubrid_get_server_info($conn);
$client_info = cubrid_get_client_info();

printf("%-30s %s\n", "Server Info:", $server_info);
printf("%-30s %s\n", "Client Info:", $client_info);

printf("\n");

$charset = cubrid_get_charset($conn);

printf("%-30s %s\n", "CUBRID Charset:", $charset);

cubrid_disconnect($conn);
?>

The above example will output:
CUBRID PHP Version:            8.3.1.0005

PARAM_ISOLATION_LEVEL          3
LOCK_TIMEOUT                   -1
MAX_STRING_LENGTH              1073741823
PARAM_AUTO_COMMIT              0

Server Info:                   8.3.1.0173
Client Info:                   8.3.1

CUBRID Charset:                iso8859-1
```

See Also

- cubrid_connect

cubrid_drop

Description

The **cubrid_drop**() function is used to drop the desired instance from the database by using the OID.

Syntax

```
bool cubrid_drop (resource $conn_identifier, string $oid)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the instance to be deleted

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");
```

```
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(2, {4,5,7}, {44,55,66,666},
'b')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

printf("--- Before Drop: ---\n");
$attr = cubrid_get($conn, $oid);
var_dump($attr);

if (cubrid_drop($conn, $oid)) {
    cubrid_commit($conn);
} else {
    cubrid_rollback($conn);
}

cubrid_close_request($req);

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

printf("\n--- After Drop: ---\n");
$attr = cubrid_get($conn, $oid);
var_dump($attr);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

--- Before Drop: ---
array(4) {
    ["a"]=>
    string(1) "1"
    ["b"]=>
    array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=> string(1) "3"
    }
    ["c"]=>
    array(4) {
    [0]=>
    string(2) "11"
    [1]=> string(2) "22"
    [2]=> string(2) "33"
    [3]=>
    string(3) "333"
    }
    ["d"]=>
    string(10) "a "
    }

--- After Drop: ---
array(4) {
    ["a"]=>
    string(1) "2"
    ["b"]=>
    array(3) {
    [0]=> string(1) "4"
    [1]=> string(1) "5"
    [2]=> string(1) "7"
    }
    ["c"]=>
    array(4) {
    [0]=>
```

```
    string(2) "44"
    [1]=>
    string(2) "55"
    [2]=> string(2) "66"
    [3]=> string(3) "666"
    } ["d"]=>
    string(10) "b "
}
```

See Also

- cubrid_is_instance

cubrid_errno, cubrid_error_code

Description

The **cubrid_errno**() function or the **cubrid_error_code**() function is used to get the code of the error that occurred during the API execution. Usually, the error message can be fetched when the API returns **FALSE**.

Syntax

```
int cubrid_errno ()
int cubrid_error_code ()
```

Return Value

- Error code

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = cubrid_prepare($conn , "SELECT * FROM code WHERE s_name=?");

$req = @cubrid_execute($req);
if (!$req) {
    printf("Error facility: %d\nError code: %d\nError msg: %s\n",
        cubrid_error_code_facility(), cubrid_error_code(), cubrid_error_msg());

    cubrid_disconnect($conn);
    exit;
}
?>

The above example will output:

Error facility: 4
Error code: -2015
Error msg: Some parameter not binded
```

See Also

- cubrid_error_code_facility
- cubrid_error_msg

cubrid_error, cubrid_error_msg

Description

The **cubrid_error**() function or the **cubrid_error_msg**() function is used to get the error message that occurred during the API execution. Usually, the error message can be fetched when the API returns **FALSE**.

Syntax

```
string cubrid_error ()
```

```
string cubrid_error_msg ()
```

Return Value

- Occurred error message

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

if (!@cubrid_schema($conn, 100000)) {
    printf("Error facility: %d\nError code: %d\nError msg: %s\n",
        cubrid_error_code_facility(), cubrid_error_code(), cubrid_error_msg());

    cubrid_disconnect($conn);
    exit;
}
?>

The above example will output:

Error facility: 2
Error code: -1015
Error msg: Invalid T_CCI_SCH_TYPE value
```

See Also

- cubrid_error_code
- cubrid_error_code_facility

cubrid_error_code_facility

Description

The **cubrid_error_code_facility**() function is used to get a facility code (level at which the error occurred) from the code of the error that occurred during the API execution. Usually, the error code can be fetched when the API returns **FALSE**.

Syntax

```
int cubrid_error_code_facility ()
```

Return Value

- Facility code of the occurred error code :
  CUBRID_FACILITY_DBMS, CUBRID_FACILITY_CAS,
  CUBRID_FACILITY_CCI, CUBRID_FACILITY_CLIENT

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = @cubrid_execute($conn, "SELECT * FROM unknown");
if (!$req) {
    printf("Error facility: %d\nError code: %d\nError msg: %s\n",
        cubrid_error_code_facility(), cubrid_error_code(), cubrid_error_msg());

    cubrid_disconnect($conn);
    exit;
}
?>

The above example will output:

Error facility: 1
Error code: -493
Error msg: Syntax: syntax error, unexpected UNKNOWN
```

See Also

- cubrid_error_code
- cubrid_error_msg

cubrid_execute

Description

The **cubrid_execute**() function is used to execute a given SQL statement. It executes a query by using *conn_identifier* and SQL and then returns the request identifier created. This is an appropriate way to simply execute a query when parameter binding is not necessary.

The **cubrid_execute**() function is also used when executing **Prepared Statement** with cubrid_prepare and cubrid_bind. In this case, required parameters are *req_identifier* and *option*.

The *option* parameter is used to determine whether to get OID after query execution and whether to execute the query in synchronous or asynchronous mode. **CUBRID_INCLUDE_OID** and **CUBRID_ASYNC** (or **CUBRID_EXEC_QUERY_ALL** if you want to execute multiple SQL statements) can be specified by using a bitwise OR operator (|). If not specified, neither of them is selected.

If the flag **CUBRID_EXEC_QUERY_ALL** is set, a synchronous mode (sync_mode) is used to retrieve query results and in such case the following rules are applied.

- The return value is the result of the first query.
- If an error occurs in any query, the execution is processed as a failure.
- For a query composed of in a query composed of q1 q2 q3 if an error occurs in q2 after q1 succeeds the execution, the result of q1 remains valid. That is, the previous successful query executions are not rolled back when an error occurs.
- If a query is executed successfully, the result of the second query can be obtained using cubrid_next_result().

If *req_identifier* is the first argument for the execution of cubrid_prepare(), only **CUBRID_ASYNC** or **CUBRID_EXEC_QUERY_ALL** can be used as an option.

Syntax

```
resource cubrid_execute (resource $conn_identifier, string $SQL [, int $option])
```

- *conn_identifier* : Connection identifier
- *SQL* : SQL statement to be executed
- *option* : Query execution option - **CUBRID_INCLUDE_OID**, **CUBRID_ASYNC**, **CUBRID_EXEC_QUERY_ALL**

```
bool cubrid_execute (resource &req_identifier[, int $option])
```

- *req_identifier* : request identifier
- *option* : Query execution option - **CUBBRID_ASYNC**, **CUBRID_EXEC_QUERY_ALL**

Return Value

- Success
- Request identifier : When process is successful and first parameter is conn_identifier
- TRUE : When process is successful and first argument is req_identifier
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$result = cubrid_execute($conn, "SELECT code FROM event WHERE name='100m Butterfly' and
gender='M'", CUBRID_ASYNC);
$row = cubrid_fetch_array($result, CUBRID_ASSOC);
$event_code = $row["code"];
```

```
cubrid_close_request($result);

$history_req = cubrid_prepare($conn, "SELECT * FROM history WHERE event_code=?");
cubrid_bind($history_req, 1, $event_code, "number");
cubrid_execute($history_req);

printf("%-20s %-9s %-10s %-5s\n", "athlete", "host_year", "score", "unit");
while ($row = cubrid_fetch_array($history_req, CUBRID_ASSOC)) {
    printf("%-20s %-9s %-10s %-5s\n",
        $row["athlete"], $row["host_year"], $row["score"], $row["unit"]);
}

cubrid_close_request($history_req);

cubrid_disconnect($conn);
?>

The above example will output:

athlete          host_year score          unit
Phelps Michael   2004      51.25          time
```

See Also

- cubrid_close_request
- cubrid_commit
- cubrid_rollback
- cubrid_prepare
- cubrid_bind

cubrid_fetch

Description

The **cubrid_fetch**() function is used to fetch one row from the query result. After the fetch, the cursor automatically moves to the next row.

Syntax

```
mixed cubrid_fetch (resource &result [, int &type])
```

- *result* : Result that comes from a call to cubrid_execute()
- *type* : Type of the result array to be fetched. CUBRID_NUM, CUBRID_ASSOC, CUBRID_BOTH, CUBRID_OBJECT

Return Value

- Success : Result array or object.

  It is determined by the *type* parameter. If the *type* parameter is omitted, **CUBRID_BOTH** is used. If you want to get the query result as an object data type, column names must comply with identifier name rules allowed in PHP. For example, a column name "count(*)" cannot be fetched and used as an object type.

  The following are different result types depending on *type*.

- CUBRID_NUM : Numeric array (0-default)
- CUBRID_ASSOC : Associative array
- CUBRID_BOTH : Numeric and associative arrays (default value)
- CUBRID_OBJECT : An object that has the attribute whose name is the same as the column name of the query result
- Failure or the end is reached : FALSE

Example

```
<?php
```

```
$conn = cubrid_connect("localhost", 33088, "demodb");
$req = cubrid_execute($conn, "SELECT * FROM stadium WHERE nation_code='GRE' AND seats >
10000");

printf("%-40s %-10s %-6s %-20s\n", "name", "area", "seats", "address");
while ($row = cubrid_fetch($req)) {
    printf("%-40s %-10s %-6s %-20s\n",
        $row["name"], $row["area"], $row["seats"], $row["address"]);
}

cubrid_close_request($req);

cubrid_disconnect($conn);
?>

The above example will output:

name                                 area            seats            address
Panathinaiko Stadium                 86300.00        50000            Athens, Greece
Olympic Stadium                      54700.00        13000            Athens, Greece
Olympic Indoor Hall                  34100.00        18800            Athens, Greece
Olympic Hall                         52400.00        21000            Athens, Greece
Olympic Aquatic Centre               42500.00        11500            Athens, Greece
Markopoulo Olympic Equestrian Centre 64000.00        15000            Markopoulo, Athens,
Greece
Faliro Coastal Zone Olympic Complex  34650.00        12171            Faliro, Athens,
Greece
Athens Olympic Stadium               120400.00       71030            Maroussi, Athens,
Greece
Ano Liossia                          34000.00        12000            Ano Liosia, Athens,
Greece
```

See Also

- cubrid_execute

cubrid_fetch_array

Description

The **cubrid_fetch_array**() function is used to get a single row from the query result and returns an array. The cursor automatically moves to the next row after getting the result.

Syntax

```
array cubrid_fetch_array(resource $result[, int $type = CUBRID_BOTH])
```

- *result* : Result that comes from a call to cubrid_execute()
- *type* : Type of the result array to be fetched. CUBRID_NUM, CUBRID_ASSOC, CUBRID_BOTH

Return Value

- Success : Returns an array of strings that corresponds to the fetched row, when process is successful.
  The type of returned array depends on how type is defined. By using **CUBRID_BOTH** (default), you'll get an array with both associative and number indices, and you can decide which data type to use by setting the type argument.
  The type variable can be set to one of the following values:
- CUBRID_NUM : Numeric array (0-based)
- CUBRID_ASSOC : Associative array
- CUBRID_BOTH : Numeric and associative arrays (default value)
- Failure or the end is reached : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = cubrid_execute($conn, "SELECT name,area,seats,address FROM stadium WHERE
nation_code='GRE' AND seats > 10000");
```

```
printf("%-40s %-10s %-6s %-20s\n", "name", "area", "seats", "address");
while ($row = cubrid_fetch_array($req, CUBRID_NUM)) {
    printf("%-40s %-10s %-6s %-20s\n", $row[0], $row[1], $row[2], $row[3]);
}

cubrid_close_request($req);

cubrid_disconnect($conn);
?>

The above example will output:

name a                                  rea        seats         address
Panathinaiko Stadium                    86300.00   50000         Athens, Greece
Olympic Stadium                         54700.00   13000         Athens, Greece
Olympic Indoor Hall                     34100.00   18800         Athens, Greece
Olympic Hall                            52400.00   21000         Athens, Greece
Olympic Aquatic Centre                  42500.00   11500         Athens, Greece
Markopoulo Olympic Equestrian Centre    64000.00   15000         Markopoulo, Athens,
Greece
Faliro Coastal Zone Olympic Complex     34650.00   12171         Faliro, Athens,
Greece Athens
Olympic Stadium                         120400.00  71030         Maroussi, Athens,
Greece
Ano Liossia                             34000.00   12000         Ano Liosia, Athens,
Greece
```

See Also

- cubrid_execute

cubrid_fetch_assoc

Description

The **cubrid_fetch_assoc**() function is used to returns an associative array that corresponds to the fetched row and moves the internal data pointer ahead or **FALSE** if there are no more rows.

Syntax

```
array cubrid_fetch_assoc ( resource $result )
```

- *result* : Result that comes from a call to cubrid_execute()

Return Value

- Success : Return associative array
- Failure or the end is reached : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = cubrid_execute($conn, "SELECT name,area,seats,address FROM stadium WHERE
nation_code='GRE' AND seats > 10000");

printf("%-40s %-10s %-6s %-20s\n", "name", "area", "seats", "address");
while ($row = cubrid_fetch_assoc($req)) {
    printf("%-40s %-10s %-6s %-20s\n",
        $row["name"], $row["area"], $row["seats"], $row["address"]);
}

cubrid_close_request($req);

cubrid_disconnect($conn);
?>

The above example will output:
```

```
name                                  area           seats             address
Panathinaiko Stadium                  86300.00       50000             Athens, Greece
Olympic Stadium                       54700.00       13000             Athens, Greece
Olympic Indoor Hall                   34100.00       18800             Athens, Greece
Olympic Hall                          52400.00       21000             Athens, Greece
Olympic Aquatic Centre                42500.00       11500             Athens, Greece
Markopoulo Olympic Equestrian Centre  64000.00       15000             Markopoulo, Athens,
Greece
Faliro Coastal Zone Olympic Complex   34650.00       12171             Faliro, Athens,
Greece
Athens Olympic Stadium                120400.00      71030             Maroussi, Athens,
Greece
Ano Liossia                           34000.00       12000             Ano Liosia, Athens,
Greece
```

cubrid_fetch_field

Description

The **cubrid_fetch_field**() function is used to return an object containing field information. This function can be used to obtain information about fields in the provided query result. The properties of the object are:

- name : Column name
- table : Name of the table where the column belongs
- def : Default value of the column
- max_length : Maximum length of the column
- not_null : 1 if the column cannot be **NULL**
- unique_key : 1 if the column is a unique key
- multiple_key : 1 if the column is a non-unique key
- numeric : 1 if the column is numeric
- type : The type of the column

Syntax

```
object cubrid_fetch_field ( resource $result [, int $field_offset= 0 ] )
```

- *result* : Result that comes from a call to cubrid_execute()
- *field_offset* : The numerical field offset. If the field offset is not specified, the next field that was not yet retrieved by this function is retrieved. The field_offset starts at 0.

Return Value

- Success: Object with certain properties of the specific column
- Failure or the end is reached : FALSE

Example

```
< ?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = cubrid_execute($conn, "SELECT event_code,athlete_code,nation_code,game_date FROM
game WHERE host_year=1988 and event_code=20001;");

var_dump(cubrid_fetch_row($req));

cubrid_field_seek($req, 1);
$field = cubrid_fetch_field($req);

printf("\n--- Field Properties ---\n");
printf("%-30s %s\n", "name:", $field->name);
printf("%-30s %s\n", "table:", $field->table);
printf("%-30s \"%s\"\n", "default value:", $field->def);
printf("%-30s %d\n", "max lenght:", $field->max_length);
printf("%-30s %d\n", "not null:", $field->not_null);
printf("%-30s %d\n", "unique key:", $field->unique_key);
printf("%-30s %d\n", "multiple key:", $field->multiple_key);
```

```
printf("%-30s %d\n", "numeric:", $field->numeric);
printf("%-30s %s\n", "type:", $field->type);

cubrid_close_request($req);

cubrid_disconnect($conn);
?>

The above example will output:

array(4) {
    [0]=>
    string(5) "20001"
    [1]=>
    string(5) "16681"
    [2]=>
    string(3) "KOR"
    [3]=> string(9) "1988-9-30"
}

--- Field Properties ---
name:                          athlete_code
table:                         game
default value:                 ""
max lenght:                    5
not null:                      1
unique key:                    1
multiple key:                  0
numeric:                       1
type:                          integer
```

cubrid_fetch_lengths

Description

The **cubrid_fetch_lengths**() function is used to return an array that corresponds to the lengths of each field in the last row fetched by CUBRID or **FALSE** on failure.

Syntax

```
array cubrid_fetch_lengths ( resource $result )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute().

Note If field data type is **BLOB**/**CLOB**, you should get its length by using **cubrid_lob_size**().

Return Value

- Success : Returns numeric array with the lengths.
- Failure or the end is reached : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$result = cubrid_execute($conn, "SELECT * FROM game WHERE host_year=2004 AND
nation_code='AUS' AND medal='G'");

$row = cubrid_fetch_row($result);
print_r($row);

$lens = cubrid_fetch_lengths($result);
print_r($lens);

cubrid_disconnect($conn);
?>

The above example will output:
```

```
Array
(
    [0] => 2004
    [1] => 20085
    [2] => 15118
    [3] => 30134
    [4] => AUS
    [5] => G
    [6] => 2004-8-20
)
Array
(
    [0] => 4
    [1] => 5
    [2] => 5
    [3] => 5
    [4] => 3
    [5] => 1
    [6] => 9
)
```

cubrid_fetch_object

Description

The **cubrid_fetch_object**() function is used to return the current row result set as an object, where the attributes of the object represent the names of the fields found within the result set. The cursor automatically moves to the next row after getting the result.

Syntax

```
object cubrid_fetch_object ( resource
$result[, string $class_name="stdClass" [, array $params ]])
```

- *result* : Result that comes from a call to cubrid_execute()
- *class_name* : The name of the class to instantiate. If not specified, a **stdClass** (stdClass is PHP's generic empty class that's used when casting other types to objects) object is returned.
- *params* : An optional array of parameters to pass to the constructor for class_name objects

Return Value

- Success: Returnan object.
- Failure or the end is reached : FALSE

Example

```
<?php
$conn = cubrid_connect("127.0.0.1", 33000, "demodb", "PUBLIC", "");
$res = cubrid_execute($conn, "SELECT * FROM code");

var_dump(cubrid_fetch_object($res));

class demodb_code {
    public $s_name = null;
    public $f_name = null;

    public function toString() {
        var_dump($this);
    }
}

var_dump(cubrid_fetch_object($res, "demodb_code");

class demodb_code_construct extends demodb_code {

public function __construct($s, $f) {
$this->s_name = $s;
$this->f_name = $f;
```

```
}

}

var_dump(cubrid_fetch_object($res, 'demodb_code_construct', array('s_name', 'f_name')));
var_dump(cubrid_fetch_object($res));

cubrid_close_request($res);
cubrid_disconnect($conn);
?>

Output:
object(stdClass)#1 (2) {
    ["s_name"]=>
string(1) "X"
    ["f_name"]=>
string(5) "Mixed"
}
object(demodb_code)#1 (2) {
    ["s_name"]=>
string(1) "W"
    ["f_name"]=>
string(5) "Woman"
}
object(demodb_code_construct)#1 (2) {
    ["s_name"]=>
string(6) "s_name"
    ["f_name"]=>
string(6) "f_name"
}
object(stdClass)#1 (2) {
    ["s_name"]=>
string(1) "B"
    ["f_name"]=>
string(6) "Bronze"
}
```

cubrid_fetch_row

Description

The **cubrid_fetch_row**() function is used to return a numerical array that corresponds to the fetched row and moves the internal data pointer ahead, or **FALSE** if there are no more rows.

Syntax

```
array cubrid_fetch_row ( resource $result )
```

- *result* : Result that comes from a call to cubrid_execute()

Return Value

- Success: Return an numeric array.
- Failure or the end is reached : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = cubrid_execute($conn, "SELECT name,area,seats,address FROM stadium WHERE
nation_code='GRE' AND seats > 10000");

printf("%-40s %-10s %-6s %-20s\n", "name", "area", "seats", "address");
while ($row = cubrid_fetch_row($req)) {
    printf("%-40s %-10s %-6s %-20s\n", $row[0], $row[1], $row[2], $row[3]);
}

cubrid_close_request($req);

cubrid_disconnect($conn);
```

```
?>

The above example will output:

name                                 area        seats         address
Panathinaiko Stadium                 86300.00    50000         Athens, Greece
Olympic Stadium                      54700.00    13000         Athens, Greece
Olympic Indoor Hall                  34100.00    18800         Athens, Greece
Olympic Hall                         52400.00    21000         Athens, Greece
Olympic Aquatic Centre               42500.00    11500         Athens, Greece
Markopoulo Olympic Equestrian Centre 64000.00    15000         Markopoulo, Athens, Greece
Faliro Coastal Zone Olympic Complex  34650.00    12171         Faliro, Athens, Greece
Athens Olympic Stadium               120400.00   71030         Maroussi, Athens, Greece
Ano Liossia                          34000.00    12000         Ano Liosia, Athens, Greece
```

cubrid_field_flags

Description

The **cubrid_field_flags**() function is used to return the field flags of the specified field. The flags are reported as a single word per flag separated by a single space, so that you can split the returned value using **explode**().

Syntax

```
string cubrid_field_flags ( resource $result , int $field_offset )
```

- *result* : Result that comes from a call to cubrid_execute()
- *field_offset* : The field_offset starts at 0.

Return Value

- Success: A string with flags
- Invalid field_offset value : FALSE
- SQL sentence is not SELECT : -1

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$result = cubrid_execute($conn, "SELECT * FROM game WHERE host_year=2004 AND
nation_code='AUS' AND medal='G'");

$col_num = cubrid_num_cols($result);

printf("%-30s %s\n", "Field Name", "Field Flags");
for($i = 0; $i < $col_num; $i++) {
    printf("%-30s %s\n", cubrid_field_name($result, $i), cubrid_field_flags($result, $i));
}

cubrid_disconnect($conn);
?>

The above example will output:

Field Name             Field Flags
host_year              not_null primary_key unique_key
event_code             not_null primary_key unique_key foreign_key
athlete_code           not_null primary_key unique_key foreign_key
stadium_code           not_null
nation_code
medal
game_date
```

cubrid_field_len

Description

The **cubrid_field_len**() function is used to return the length of the specified field on success, or **FALSE** on failure.

Syntax

```
string cubrid_field_len ( resource $result , int $field_offset )
```

- *result* : Result that comes from a call to cubrid_execute()
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Return Value

- Success: Maximum length
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$result = cubrid_execute($conn, "SELECT * FROM game WHERE host_year=2004 AND
nation_code='AUS' AND medal='G'");

$column_names = cubrid_column_names($result);
$column_types = cubrid_column_types($result);

printf("%-30s %-30s %-15s\n", "Column Names", "Column Types", "Column Maxlen");
for($i = 0, $size = count($column_names); $i < $size; $i++) {
    $column_len = cubrid_field_len($result, $i);
    printf("%-30s %-30s %-15s\n", $column_names[$i], $column_types[$i], $column_len);
}

cubrid_disconnect($conn);
?>

The above example will output:

Column Names                     Column Types                  Column Maxlen
host_year                        integer                       11
event_code                       integer                       11
athlete_code                     integer                       11
stadium_code                     integer                       11
nation_code                      char(3)                       3
medal                            char(1)                       1
game_date                        date                          10
```

cubrid_field_name

Description

The **cubrid_field_name**() function is used to return the name of the specified field index on success, or **FALSE** on failure.

Syntax

```
string cubrid_field_name ( resource $result , int $field_offset )
```

- *result* : Result that comes from a call to cubrid_execute()
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Return Value

- Success: Name of specified field index
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$result = cubrid_execute($conn, "SELECT * FROM game WHERE host_year=2004 AND
nation_code='AUS' AND medal='G'");
$col_num = cubrid_num_cols($result);

printf("%-30s %s\n", "Field Name", "Field Flags");
for($i = 0; $i < $col_num; $i++) {
    printf("%-30s %s\n", cubrid_field_name($result, $i), cubrid_field_flags($result, $i));
}

cubrid_disconnect($conn);
?>

The above example will output:
Field Name                     Field Flags
host_year                      not_null primary_key unique_key
event_code                     not_null primary_key unique_key foreign_key
athlete_code                   not_null primary_key unique_key foreign_key
stadium_code                   not_null
nation_code
medal
game_date
```

cubrid_field_seek

Description

The **cubrid_field_seek**() function is used to set a field offset value to be used in cubrid_fetch_field() function. If the **cubrid_fetch_field**() function that does not include a field offset is called, the field offset specified in this function is returned.

Syntax

```
bool cubrid_field_seek ( resource $result , int $field_offset )
```

- *result* : Result that comes from a call to cubrid_execute()
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$req = cubrid_execute($conn, "SELECT event_code,athlete_code,nation_code,game_date FROM
game WHERE host_year=1988 and event_code=20001;");

var_dump(cubrid_fetch_row($req));

cubrid_field_seek($req, 1);
$field = cubrid_fetch_field($req);

printf("\n--- Field Properties ---\n");
printf("%-30s %s\n", "name:", $field->name);
printf("%-30s %s\n", "table:", $field->table);
printf("%-30s \"%s\"\n", "default value:", $field->def);
printf("%-30s %d\n", "max lenght:", $field->max_length);
printf("%-30s %d\n", "not null:", $field->not_null);
printf("%-30s %d\n", "unique key:", $field->unique_key);
printf("%-30s %d\n", "multiple key:", $field->multiple_key);
printf("%-30s %d\n", "numeric:", $field->numeric);
printf("%-30s %s\n", "type:", $field->type);
```

```
cubrid_close_request($req);

cubrid_disconnect($conn);
?>

The above example will output:

array(4) {
    [0]=>
    string(5) "20001"
    [1]=>
    string(5) "16681"
    [2]=>
    string(3) "KOR"
    [3]=>
    string(9) "1988-9-30"
}

--- Field Properties ---
name:                         athlete_code
table:                        game
default value:                ""
max length:                   5
not null:                     1
unique key:                   1
multiple key:                 0
numeric:                      1
type:                         integer
```

cubrid_field_table

Description

The **cubrid_field_table**() function is used to return the name of the table that the specified field is in.

Syntax

```
string cubrid_field_table ( resource $result , int $field_offset )
```

- *result* : Result that comes from a call to cubrid_execute()
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Return Value

- Success : Name of the table of the specified field
- Invalid field_offset value : FALSE
- SQL sentence is not SELECT : -1

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$result = cubrid_execute($conn, "SELECT * FROM code");

$col_num = cubrid_num_cols($result);

printf("%-15s %-15s %s\n", "Field Table", "Field Name", "Field Type");
for($i = 0; $i < $col_num; $i++) {
    printf("%-15s %-15s %s\n",
        cubrid_field_table($result, $i), cubrid_field_name($result, $i),
cubrid_field_type($result, $i));
}

cubrid_disconnect($conn);
?>

The above example will output:
Field Table             Field Name             Field Type
code                    s_name                 char(1)
```

```
code                    f_name                 varchar(6)
```

cubrid_field_type

Description

The **cubrid_field_type**() function is similar to the **cubrid_field_name**() function. The arguments are identical, but the field type is returned instead. The returned field type will be one of "int", "real", "string", etc.

Syntax

```
string cubrid_field_type ( resource $result , int $field_offset )
```

- *result* : Result that comes from a call to cubrid_execute()
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Result Value

- Success : Type of the column
- When invalid field_offset value : FALSE
- SQL sentence is not SELECT : -1

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");
$result = cubrid_execute($conn, "SELECT * FROM code");

$col_num = cubrid_num_cols($result);

printf("%-15s %-15s %s\n", "Field Table", "Field Name", "Field Type");
for($i = 0; $i < $col_num; $i++) {
    printf("%-15s %-15s %s\n",
        cubrid_field_table($result, $i), cubrid_field_name($result, $i),
cubrid_field_type($result, $i));
}

cubrid_disconnect($conn);
?>

The above example will output:
Field Table             Field Name             Field Type
code                    s_name                 char(1)
code                    f_name                 varchar(6)
```

cubrid_free_result

Description

The **cubrid_free_result**() function is used to free the memory occupied by the result data.

Note The **cubrid_free_result**() function can only frees the client fetch buffer now, and if you want free all memory occupied by the result data, use function cubrid_close_request().

Syntax

```
bool cubrid_free_result ( resource $result )
```

- *result* : Result that comes from a call to cubrid_execute

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$req = cubrid_execute($conn, "SELECT * FROM history WHERE host_year=2004 ORDER BY
event_code");
$row = cubrid_fetch_assoc($req);
var_dump($row);

cubrid_free_result($req);
cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(5) {
    ["event_code"]=>
    string(5) "20005"
    ["athlete"]=>
    string(12) "Hayes Joanna"
    ["host_year"]=>
    string(4) "2004"
    ["score"]=>
    string(5) "12.37"
    ["unit"]=>
    string(4) "time"
}
```

cubrid_get

Description

The **cubrid_get**() function is used to get a desired attribute of an instance by using OID. You can get a single attribute by using a character string type for the *attr* argument, or multiple attributes by using an array type.

Syntax

```
mixed cubrid_get (resource $conn_identifier, string $oid[, mixed $attr])
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the instance whose value you want to get
- *attr* : Name of the attribute whose value you want to get

Return Value

A character string is returned if a character string type is set for the *attr* argument; an associative array is returned if an array type (0 - default numeric array) is set. If the *attr* argument is omitted, all attributes of the instance are returned as an associative array.

- Success : Content of the attribute(s) requested
- Failure : FALSE. If an error occurs, a warning message is displayed to distinguish it from an empty character string or **NULL**. You can check the error with cubrid_error_code().

Example

```
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(2, {4,5,7}, {44,55,66,666},
'b')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);
```

```
cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_get($conn, $oid, "b");
var_dump($attr);

$attr = cubrid_get($conn, $oid);
var_dump($attr);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

string(9) "{1, 2, 3}"
array(4) {
    ["a"]=>
    string(1) "1"
    ["b"]=>
    array(3) {
    [0]=> string(1) "1"
    [1]=> string(1)
    "2" [2]=> string(1) "3"
    }
    ["c"]=>
    array(4) {
    [0]=> string(2) "11"
    [1]=> string(2) "22"
    [2]=> string(2) "33"
    [3]=> string(3) "333"
    }
    ["d"]=>
    string(10) "a "
}
```

See Also

- cubrid_put

cubrid_get_autocommit

Description

The **cubrid_get_autocommit**() function is used to get the status of CUBRID database connection auto-commit mode.

Syntax

```
bool cubrid_get_autocommit (resource $conn_identifier)
```

- *conn_identifier* : Connection identifier

Return Value

- Auto-commit on : TRUE
- Auto-commit off : FALSE

See Also

- cubrid_set_autocommit

cubrid_get_charset

Description

The **cubrid_get_charset**() function is used to get CUBRID current connection charset.

Syntax

```
string cubrid_get_charset ( resource $req_identifier )
```

- *req_identifier* : Request identifier

Return Value

- Success : A string that represents the CUBRID connection charset
- Failure : FALSE

Example

```
<?php
printf("%-30s %s\n", "CUBRID PHP Version:", cubrid_version());

printf("\n");

$conn = cubrid_connect("localhost", 33088, "demodb");

if (!$conn) {
    die('Connect Error ('. cubrid_error_code() .')' . cubrid_error_msg());
}

$db_params = cubrid_get_db_parameter($conn);
while (list($param_name, $param_value) = each($db_params)) {
    printf("%-30s %s\n", $param_name, $param_value);
}

printf("\n");

$server_info = cubrid_get_server_info($conn);
$client_info = cubrid_get_client_info();

printf("%-30s %s\n", "Server Info:", $server_info);
printf("%-30s %s\n", "Client Info:", $client_info);

printf("\n");

$charset = cubrid_get_charset($conn);

printf("%-30s %s\n", "CUBRID Charset:", $charset);

cubrid_disconnect($conn);

?>

The above example will output:

CUBRID PHP Version:             8.3.1.0005

PARAM_ISOLATION_LEVEL           3
LOCK_TIMEOUT                    -1
MAX_STRING_LENGTH               1073741823
PARAM_AUTO_COMMIT               0

Server Info:                    8.3.1.0173
Client Info:                    8.3.1

CUBRID Charset:                 iso8859-1
```

cubrid_get_class_name

Description

The **cubrid_get_class_name**() function is used to get a class name from an OID.

Syntax

```
mixed  cubrid_is_instance (resource $conn_identifier, string $oid)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of an instance, for which you want to check whether it exists

Return Value

- Success : Class name
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33088, "demodb");

$req = cubrid_execute($conn, "SELECT * FROM code", CUBRID_INCLUDE_OID);
$oid = cubrid_current_oid($req);
$class_name = cubrid_get_class_name($conn, $oid);

print_r($class_name);

cubrid_disconnect($conn);
?>

The above example will output:
code
```

See Also

- cubrid_is_instance
- cubrid_drop

cubrid_get_client_info

Description

The **cubrid_get_client_info**() function is used to return a string that represents the client library version.

```
string cubrid_get_client_info ( void )
```

Return Value

- Success : A string that represents the client library version
- Failure : FALSE

Example

```
<?php
printf("%-30s %s\n", "CUBRID PHP Version:", cubrid_version());

printf("\n");

$conn = cubrid_connect("localhost", 33088, "demodb");

if (!$conn) {
    die('Connect Error ('. cubrid_error_code() .')' . cubrid_error_msg());
}

$db_params = cubrid_get_db_parameter($conn);

while (list($param_name, $param_value) = each($db_params)) {
    printf("%-30s %s\n", $param_name, $param_value);
}

printf("\n");

$server_info = cubrid_get_server_info($conn);
$client_info = cubrid_get_client_info();

printf("%-30s %s\n", "Server Info:", $server_info);
```

```
printf("%-30s %s\n", "Client Info:", $client_info);

printf("\n");

$charset = cubrid_get_charset($conn);

printf("%-30s %s\n", "CUBRID Charset:", $charset);
cubrid_disconnect($conn);

?>

The above example will output:

CUBRID PHP Version:             8.3.1.0005

PARAM_ISOLATION_LEVEL           3
LOCK_TIMEOUT                    -1
MAX_STRING_LENGTH               1073741823
PARAM_AUTO_COMMIT               0

Server Info:                    8.3.1.0173
Client Info:                    8.3.1

CUBRID Charset:                 iso8859-1
```

cubrid_get_db_parameter

Description

The **cubrid_get_db_parameter**() function returns the CUBRID system parameters. It returns the CUBRID system parameters or it returns FALSE on failure. It returns an associative array with the values for the following parameters:

- **PARAM_ISOLATION_LEVEL** : In CUBRID PHP, you can set the level of transaction isolation by using cubrid_set_db_parameter() function, isolation_level in the $**CUBRID/conf/cubrid.conf** or the **SET TRANSACTION** statement. For levels of isolation supported by CUBRID, refer SET TRANSACTION ISOLATION LEVEL on the manual.
- **PARAM_LOCK_TIMEOUT** : CUBRID provides the lock timeout feature, which sets the waiting time for the lock until the transaction lock setting is allowed. You can set lock timeout by using cubrid_set_db_parameter() function, parameter lock_timeout_in_secs in the $**CUBRID/conf/cubrid.conf** file or the SET TRANSACTION statement (in seconds). The default value of the lock_timeout_in_secs parameter is -1, which means the application client will wait indefinitely until the transaction lock is allowed.

PARAM_MAX_STRING_LENGTH : The maximum string length of a parameter

- **PARAM_AUTO_COMMIT** : In CUBRID PHP, an auto-commit mode is enabled by default for transaction management. If you want to start a transaction, you should set auto-commit mode to off by using the cubrid_set_autocommit() function. And auto commit modes can be applied only for SELECT statements by setting broker parameters.

Syntax

```
array cubrid_get_db_parameter ( resource $conn_identifier )
```

- *conn_identifier* : Connection identifier previously obtained from a call to cubrid_connect()

Return Value

- Success : An associative array with CUBRID system parameters
- Failure : FALSE

Example

```
<?php
printf("%-30s %s\n", "CUBRID PHP Version:", cubrid_version());

printf("\n");

$conn = cubrid_connect("localhost", 33088, "demodb");
```

```
if (!$conn) {
    die('Connect Error ('. cubrid_error_code() .')' . cubrid_error_msg());
}

$db_params = cubrid_get_db_parameter($conn);

while (list($param_name, $param_value) = each($db_params)) {
    printf("%-30s %s\n", $param_name, $param_value);
}

printf("\n");

$server_info = cubrid_get_server_info($conn);
$client_info = cubrid_get_client_info();

printf("%-30s %s\n", "Server Info:", $server_info);
printf("%-30s %s\n", "Client Info:", $client_info);
printf("\n");

$charset = cubrid_get_charset($conn);

printf("%-30s %s\n", "CUBRID Charset:", $charset);

cubrid_disconnect($conn);

?>

The above example will output:

CUBRID PHP Version:                    8.3.1.0005

PARAM_ISOLATION_LEVEL                  3
LOCK_TIMEOUT                           -1
MAX_STRING_LENGTH                      1073741823
PARAM_AUTO_COMMIT                      0

Server Info:                           8.3.1.0173
Client Info:                           8.3.1

CUBRID Charset:                        iso8859-1
```

See Also

- cubrid_set_db_parameter

cubrid_get_server_info

Description

The **cubrid_get_server_info**() function returns a string that represents the CUBRID server version.

Syntax

```
string cubrid_get_server_info ( resource $conn_identifier )
```

- *conn_identifier* : Connection identifier previously obtained from a call to cubrid_connect()

Return Value

- Success : A string that represents the CUBRID server version
- Failure : FALSE

Examples

```
<?php
printf("%-30s %s\n", "CUBRID PHP Version:", cubrid_version());

printf("\n");

$conn = cubrid_connect("localhost", 33088, "demodb");
```

```
if (!$conn) {
    die('Connect Error ('. cubrid_error_code() .')' . cubrid_error_msg());
}

$db_params = cubrid_get_db_parameter($conn);

while (list($param_name, $param_value) = each($db_params)) {
    printf("%-30s %s\n", $param_name, $param_value);
}

printf("\n");

$server_info = cubrid_get_server_info($conn);
$client_info = cubrid_get_client_info();

printf("%-30s %s\n", "Server Info:", $server_info);
printf("%-30s %s\n", "Client Info:", $client_info);
printf("\n");

$charset = cubrid_get_charset($conn);

printf("%-30s %s\n", "CUBRID Charset:", $charset);

cubrid_disconnect($conn);

?>

The above example will output:

CUBRID PHP Version:                    8.3.1.0005

PARAM_ISOLATION_LEVEL                  3
LOCK_TIMEOUT                           -1
MAX_STRING_LENGTH                      1073741823
PARAM_AUTO_COMMIT                      0

Server Info:                           8.3.1.0173
Client Info:                           8.3.1

CUBRID Charset:                        iso8859-1
```

cubrid_insert_id

Description

The **cubrid_insert_id**() function retrieves the ID generated for the **AUTO_INCREMENT** columns which is updated by the previous **INSERT** query. It returns 0 if the previous query does not generate new rows, or **FALSE** on failure.

Note CUBRID supports AUTO_INCREMENT for more than one column in a table. In most cases, there will be a single **AUTO_INCREMENT** column in a table. If there are multiple **AUTO_INCREMENT** columns, the **cubrid_insert_id**() should not be used even if it will return a value.

Syntax

```
array cubrid_insert_id ( string $class_name [, resource $conn_identifier] )
```

- *class_name* : The name of the class (table) that was used in the last **INSERT** statement for which the auto increment values are retrieved.
- *connection_identifier* : Connection identifier previously obtained from a call to cubrid_connect()

Return Value

- Success : A string representing the ID generated for **AUTO_INCREMENT** column by the previous query
- If the previous query does not generate new rows : 0
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE cubrid_test");
cubrid_execute($conn, "CREATE TABLE cubrid_test (d int AUTO_INCREMENT(1, 2), t varchar)");

for ($i = 0; $i < 10; $i++) {
    cubrid_execute($conn, "INSERT INTO cubrid_test(t) VALUES('cubrid_test')");
}

$id_list = cubrid_insert_id("cubrid_test");
var_dump($id_list);

cubrid_disconnect($conn);
?>

The above example will output:

array(1) {
    ["d"]=>
    int(19)
}
```

cubrid_is_instance

Description

The **cubrid_is_instance**() function is used to check whether an instance referred to by an OID exists in the database.

Syntax

```
int cubrid_is_instance (resource $conn_identifier, string $oid)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of an instance, for which you want to check whether it exists

Return Value

- An instance exists : 1
- An instance does not exist : 0
- An error occurs : -1

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$sql = <<EOD
SELECT host_year, medal, game_date
FROM game
WHERE athlete_code IN
    (SELECT code FROM athlete WHERE name='Thorpe Ian');
EOD;

$req = cubrid_execute($conn, $sql, CUBRID_INCLUDE_OID);
$oid = cubrid_current_oid($req);

$res = cubrid_is_instance ($conn, $oid);
if ($res == 1) {
    echo "Instance pointed by $oid exists.\n";
} else if ($res == 0){
    echo "Instance pointed by $oid doesn't exist.\n";
} else {
    echo "error\n";
}

cubrid_disconnect($conn);
```

```
?>

The above example will output:
Instance pointed by @0|0|0 doesn't exist.
```

See Also

- cubrid_drop
- cubrid_get_class_name

cubrid_lob_close

Description

The **cubrid_lob_close**() function is used to close **BLOB**/**CLOB** returned from cubrid_lob_get().

Syntax

```
bool cubrid_lob_close (array $lob_identifier_array)
```

- *lob_identifier_array* : LOB identifier array returned from cubrid_lob_get()

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$lobs = cubrid_lob_get($con, "SELECT doc_content FROM doc WHERE doc_id=5");
cubrid_lob_export($conn, $lobs[0], "doc_5.txt");
cubrid_lob_close($lobs);
?>
```

See Also

- cubrid_lob_get
- cubrid_lob_send
- cubrid_lob_export
- cubrid_lob_size

cubrid_lob_export

Description

The **cubrid_lob_export**() function is used to export **BLOB**/**CLOB** data to file.

Syntax

```
bool cubrid_lob_export(resource $conn_identifier, resource $lob_identifier, string
$path_name )
```

- *conn_identifier* : Connection identifier
- *lob_identifier* : LOB identifier
- *path_name* : Path name of file

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$lobs = cubrid_lob_get($con, "SELECT doc_content FROM doc WHERE doc_id=5");
cubrid_lob_export($conn, $lobs[0], "doc_5.txt");
cubrid_lob_close($lobs);
?>
```

See Also

- cubrid_lob_get
- cubrid_lob_send
- cubrid_lob_export
- cubrid_lob_close

cubrid_lob_get

Description

The **cubrid_lob_get**() function is used to get **BLOB**/**CLOB** meta info from CUBRID database. CUBRID gets **BLOB**/**CLOB** by executing a SQL statement, and returns all LOBs as a resource array. Be sure that the SQL retrieves only one column and its data type is **BLOB** or **CLOB**.

Remember to use cubrid_lob_close() to release the LOBs if you don't need it any more.

Syntax

```
bool cubrid_lob_get(resource $conn_identifier, string $SQL )
```

- *conn_identifier* : Connection identifier
- *SQL* : SQL statement to be executed.

Return Value

- Success : An array of LOB resources
- Failure : FALSE

Example

```
<?php
$lobs = cubrid_lob_get($con, "SELECT doc_content FROM doc WHERE doc_id=5");
cubrid_lob_export($conn, $lobs[0], "doc_5.txt");
cubrid_lob_close($lobs);
?>
```

See Also

- cubrid_lob_close
- cubrid_lob_export
- cubrid_lob_send
- cubrid_lob_size

cubrid_lob_send

Description

The **cubrid_lob_send**() function reads **BLOB**/**CLOB** data and passes it straight through to the browser. To use this function, you must use cubrid_lob_get() first to get **BLOB**/**CLOB** info from CUBRID.

Syntax

```
bool cubrid_lob_send(resource $conn_identifier, resource $lob_identifier )
```

- *conn_identifier* : Connection identifier
- *lob_identifier* : LOB identifier

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$lobs = cubrid_lob_get($con, "SELECT image FROM doc WHERE id=1");
Header("Content-type: image/jpeg");
cubrid_lob_send($conn, $lobs[0]);
cubrid_lob_close($lobs);
?>
```

See Also

- cubrid_lob_close
- cubrid_lob_export
- cubrid_lob_get
- cubrid_lob_size

cubrid_lob_size

Description

The **cubrid_lob_size**() function is used to get **BLOB**/**CLOB** data size.

Syntax

```
bool cubrid_lob_size(resource $lob_identifier )
```

- *lob_identifier* : LOB identifier

Note The maximum length of **BLOB**/**CLOB** data is the maximum file size creatable in an external storage. The type of LOB size in CUBRID PHP is 64-bit integer, but it can’t return 64-bit integer type in PHP, so returns a string instead.

Return Value

- Success : LOB data size, as a string
- Failure : FALSE

Example

```
<?php
$lobs = cubrid_lob_get($con, "SELECT doc_content FROM doc WHERE doc_id=5");
echo "Doc size:".cubrid_lob_size($lobs[0]);
cubrid_lob_export($conn, $lobs[0], "doc_5.txt");
cubrid_lob_close($lobs);
?>
```

See Also

- cubrid_lob_close
- cubrid_lob_export
- cubrid_lob_get
- cubrid_lob_send

cubrid_list_dbs

Description

The **cubrid_list_dbs**() function returns an array with the list of all existing CUBRID database.

Syntax

```
array cubrid_list_dbs (resource $conn_identifier)
```

- *conn_identifier* : Connection identifier previously obtained from a call to cubrid_connect()

Example

```
<?php
$conn = cubrid_connect("localhost", 33088, "demodb");

$db_list = cubrid_list_dbs($conn);
var_dump($db_list);

cubrid_disconnect($conn);
?>
The above example will output:

array(1) {
    [0]=>
    string(6) "demodb"
}
```

See Also

- cubrid_db_name

cubrid_lock_read

Description

The **cubrid_lock_read**() function is used to configure a read lock on the given instance by using an OID.

Syntax

```
bool cubrid_lock_read (resource $conn_identifier, string $oid)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of an instance on which you want to configure a lock

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33088, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(2, {4,5,7}, {44,55,66,666},
'b')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);
```

```
cubrid_lock_read($conn, $oid);

$attr = cubrid_get($conn, $oid, "b");
var_dump($attr);

$attr = cubrid_get($conn, $oid);
var_dump($attr); cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:
string(9) "{1, 2, 3}"
array(4) {
    ["a"]=>
    string(1) "1"
    ["b"]=>
    array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "3"
    }
    ["c"]=>
    array(4) {
    [0]=>
    string(2) "11"
    [1]=>
    string(2) "22"
    [2]=>
    string(2) "33"
    [3]=> string(3) "333"
    }
    ["d"]=>
    string(10) "a "
}
```

See Also

- cubrid_lock_write

cubrid_lock_write

Description

The **cubrid_lock_write**() function is used to configure a write lock on the given instance using an OID.

Syntax

```
bool cubrid_lock_write (resource $conn_identifier, string $oid)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of an instance on which you want to configure a lock

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33088, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
```

```
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(2, {4,5,7}, {44,55,66,666},
'b')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

cubrid_lock_read($conn, $oid);

$attr = cubrid_get($conn, $oid, "b");
var_dump($attr);

$attr = cubrid_get($conn, $oid);
var_dump($attr); cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(3) {
   [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "3"
}
array(3) {
    [0]=>
    string(1) "2"
    [1]=>
    string(1) "4"
    [2]=>
    string(1) "8"
}
```

See Also

- cubrid_lock_read

cubrid_move_cursor

Description

The **cubrid_move_cursor**() function is used to move the current cursor position of *req_identifier* to the distance configured by the offset argument in the direction in the origin argument. For origin, the first position in the result (**CUBRID_CURSOR_FIRST**), the current position in the result (**CUBRID_CURSOR_CURRENT**) and the last position in the result (**CUBRID_CURSOR_LAST**) can be used. If origin is not specified, **CUBRID_CURSOR_CURRENT** is used by default.

If the amount of cursor movement exceeds the range of the result, the cursor moves to a position next to the end of the result range. For example, if the cursor moves to the position 20 when the size of the result is 10, it moves to the 11th position and returns **CUBRID_NO_MORE_DATA**.

Syntax

```
int cubrid_move_cursor (resource $req_identifier, int $offset[, int $origin])
```

- *req_identifier* : Request identifier
- *offset* : The number of positions to which the cursor is to be moved
- *origin* : Origin of the cursor movement CUBRID_CURSOR_FIRST, CUBRID_CURSOR_CURRENT, CUBRID_CURSOR_LAST

Return Value

- Success : CUBRID_CURSOR_SUCCESS
- No more data : CUBRID_NO_MORE_DATA
- Failure : CUBRID_CURSOR_ERROR

Example

```
<?php
$conn = cubrid_connect("127.0.0.1", 33000, "demodb");

$req = cubrid_execute($conn, "SELECT * FROM code");
cubrid_move_cursor($req, 1, CUBRID_CURSOR_LAST);

$result = cubrid_fetch_row($req);
var_dump($result);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$result = cubrid_fetch_row($req);
var_dump($result);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_CURRENT);
$result = cubrid_fetch_row($req);
var_dump($result);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(2) {
    [0]=>
    string(1) "G"
    [1]=> string(4) "Gold"
}
    array(2) {
    [0]=>
    string(1) "X"
    [1]=> string(5) "Mixed"
}
    array(2) {
    [0]=>
    string(1) "M"
    [1]=>
    string(3) "Man"
}
```

See Also

- cubrid_execute

cubrid_next_result

Description

The **cubrid_next_result**() function is used to get results of next query if CUBRID_EXEC_QUERY_ALL flag is set upon cubrid_execute(). If next result is executed successfully, the database is updated with the information of the current query.

Syntax

```
bool cubrid_next_result (resource $result)
```

- *result* : Result that comes from a call to cubrid_execute()

Return Value

- Success : TRUE
- Failure or no more result : FALSE

Example

```
<?php
$conn = cubrid_connect($host, $port, $db, $user, $passwd);

$sql_stmt = "SELECT * FROM code; SELECT * FROM history WHERE host_year=2004 AND
event_code=20281";
$res = cubrid_execute($conn, $sql_stmt, CUBRID_EXEC_QUERY_ALL);

get_result_info($res);

cubrid_next_result($res);

get_result_info($res);

function get_result_info($req)
{
    printf("\n------------ get_result_info --------------------\n");

    $row_num = cubrid_num_rows($req);
    $col_num = cubrid_num_cols($req);

    $column_name_list = cubrid_column_names($req);
    $column_type_list = cubrid_column_types($req);

    $column_last_name = cubrid_field_name($req, $col_num - 1);
$column_last_table = cubrid_field_table($req, $col_num - 1);

    $column_last_type = cubrid_field_type($req, $col_num - 1);
    $column_last_len = cubrid_field_len($req, $col_num - 1);

    $column_1_flags = cubrid_field_flags($req, 1);

    printf("%-30s %d\n", "Row count:", $row_num);
    printf("%-30s %d\n", "Column count:", $col_num);
    printf("\n");

    printf("%-30s %-30s %-15s\n", "Column Names", "Column Types", "Column Len");
    printf("------------------------------------------------------------------------------
\n");
    $size = count($column_name_list);
    for($i = 0; $i < $size; $i++) {
    $column_len = cubrid_field_len($req, $i);
    printf("%-30s %-30s %-15s\n", $column_name_list[$i], $column_type_list[$i],
$column_len);
    }
    printf("\n\n");

    printf("%-30s %s\n", "Last Column Name:", $column_last_name);
    printf("%-30s %s\n", "Last Column Table:", $column_last_table);
    printf("%-30s %s\n", "Last Column Type:", $column_last_type);
    printf("%-30s %d\n", "Last Column Len:", $column_last_len);
    printf("%-30s %s\n", "Second Column Flags:", $column_1_flags);

    printf("\n\n");
}
?>

The above example will output:


------------ get_result_info --------------------
Row count:                     6
Column count:                  2

Column Names                   Column Types                   Column Len
------------------------------------------------------------------------------
```

```
s_name                         char(1)                        1
f_name                         varchar(6)                     6


Last Column Name:              f_name
Last Column Table:             code
Last Column Type:              varchar(6)
Last Column Len:               6
Second Column Flags:



------------ get_result_info --------------------
Row count:                     4
Column count:                  5

Column Names                   Column Types                   Column Len
------------------------------------------------------------------------------
event_code                     integer                        11
athlete                        varchar(40)                    40
host_year                      integer                        11
score                          varchar(10)                    10
unit                           varchar(5)                     5


Last Column Name:              unit
Last Column Table:             history
Last Column Type:              varchar(5)
Last Column Len:               5
Second Column Flags:           not_null primary_key unique_key
```

See Also

- cubrid_execute

cubrid_num_cols, cubrid_num_fields

Description

The **cubrid_num_cols**() function or the **cubrid_num_fields**() function is used to return the number of columns in the query result. This method is available only with the **SELECT** statement.

Syntax

```
int cubrid_num_cols (resource $req_identifier)
int cubrid_num_fields (resource $req_identifier)
```

- *req_identifier* : Request identifier

Return Value

- Success : The number of columns
- Error occurs : -1

Example

```
$req = cubrid_execute ($con, "select * from member");
if ($req) {
    $rows_count = cubrid_num_rows ($req);
    $cols_count = cubrid_num_cols ($req);
    echo "result set rows count : $rows\n";
    echo "result set columns count : $cols\n";
    cubrid_close_request ($req);
}
```

See Also

- cubrid_execute

- cubrid_num_rows

cubrid_num_rows

Description

The **cubrid_num_rows**() function is used to return the number of rows in the query result. This is available only with the **SELECT** statement. Use cubrid_affected_rows() if you want to know the results of **INSERT**, **UPDATE** and **DELETE** queries. **cubrid_num_rows**() can be used only with synchronous queries. It returns 0 if the query is asynchronous.

Syntax

```
int cubrid_num_rows (resource $request)
```

- *request* : Request that comes from a call to cubrid_execute()

Return Value

- Success : The number of rows
- Asynchronous query : 0
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$req = cubrid_execute($conn, "SELECT * FROM code");

$row_num = cubrid_num_rows($req);
$col_num = cubrid_num_cols($req);

printf("Row Num: %d\nColumn Num: %d\n", $row_num, $col_num);

cubrid_disconnect($conn);
?>

The above example will output:

Row Num: 6
Column Num: 2
```

See Also

- cubrid_execute
- cubrid_num_cols
- cubrid_affected_rows

cubrid_ping

Description

The **cubrid_ping**() function pings a server connection or reconnection if there is no connection.

Syntax

```
bool cubrid_ping ([resource $conn_identifier ])
```

- *conn_identifier* : Connection identifier. If the connection identifier is not specified, the last connection is assumed.

Return Value

- If the connection to the database server is working : TRUE

- Otherwise : FALSE

Example

```
<?php
set_time_limit(0);

$conn = cubrid_connect('localhost', 33000, 'demodb');

/* Assuming this query will take a long time */
$result = cubrid_query($sql);
if (!$result) {
echo 'Query #1 failed, exiting.';
exit;
}

/* Make sure the connection is still alive, if not, try to reconnect */
if (!cubrid_ping($conn)) {
echo 'Lost connection, exiting after query #1';
exit;
}
cubrid_free_result($result);

/* So the connection is still alive, let's run another query */
$result2 = cubrid_query($sql2);
?>
```

cubrid_prepare

Description

The **cubrid_prepare**() function is an API that represents a precompiled SQL statement on the given connection handle. The SQL statement is pre-compiled and then included in **cubrid_prepare**(). This method can be used to efficiently execute the statement multiple times or to effectively process Long Data. You can use only a single statement and a parameter can insert a question mark (?) into appropriate position in the SQL statement. You can also add a parameter to the position in the **VALUES** clause of the **INSERT** statement or in the **WHERE** clause of the SQL statement, for which the value is to be substituted. Substituting a value for a question mark (?) can be performed only by cubrid_bind.

Syntax

```
resource cubrid_prepare (resource $conn_identifier, string $prepare_stmt [, int $option])
```

- *conn_identifier* : Connection handle
- *prepare_stmt* : A prepare query
- *option* : OID return option - CUBRID_INCLUDE_OID

Return Value

- Success : Request handle
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$sql = <<EOD
SELECT g.event_code, e.name
FROM game g
JOIN event e ON g.event_code=e.code
WHERE host_year = ? AND event_code NOT IN (SELECT event_code FROM game WHERE host_year=?)
GROUP BY event_code;
EOD;

$req = cubrid_prepare($conn, $sql);

cubrid_bind($req, 1, 2004);
```

```
cubrid_bind($req, 2, 2000);
cubrid_execute($req);

$row_num = cubrid_num_rows($req);
printf("There are %d event that exits in 2004 olympic but not in 2000. For example:\n\n",
$row_num);

printf("%-15s %s\n", "Event_code", "Event_name");
printf("----------------------------\n");

$row = cubrid_fetch_assoc($req);
printf("%-15d %s\n", $row["event_code"], $row["name"]);
$row = cubrid_fetch_assoc($req);
printf("%-15d %s\n", $row["event_code"], $row["name"]);

cubrid_disconnect($conn);
?>

The above example will output:

There are 27 event that exits in 2004 olympic but not in 2000. For example:

Event_code         Event_name
----------------------------
20063              +91kg
20070              64kg
```

See Also

- cubrid_execute
- cubrid_bind

cubrid_put

Description

The **cubrid_put**() function is used to change attribute values of an instance by using the given OID. You can update single attribute by using string data type to set *attr*. In such case, you can use integer, floating-point, or character string data type for the value argument. To change multiple attributes simultaneously, pass value argument in the form of associative array data type without specifying the *attr* argument.

Syntax

```
int cubrid_put (resource $conn_identifier, string $oid[, string $attr], mixed $value)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the instance whose value you want to change
- *attr* : Name of the attribute whose value you want to change
- *value* : Value of the attribute you want to change

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(2, {4,5,7}, {44,55,66,666},
'b')");
```

```
$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_col_get($conn, $oid, "b");
var_dump($attr);

cubrid_put($conn, $oid, "b", array(2, 4, 8));

$attr = cubrid_col_get($conn, $oid, "b");
var_dump($attr);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "3"
}
array(3) {
    [0]=>
    string(1) "2"
    [1]=> string(1) "4"
    [2]=>
    string(1) "8"
}
```

See Also

- cubrid_get
- cubrid_set_add
- cubrid_set_drop
- cubrid_seq_insert
- cubrid_seq_drop
- cubrid_seq_put

cubrid_query

Description

The **cubrid_query**() function sends a unique query (multiple queries are supported) to the currently active database on the server that's associated with the specified *conn_identifier*.

The returned result resource should be passed to functions for dealing with result tables, to access the returned data. Use cubrid_num_rows() to find out how many rows were returned for a SELECT statement or cubrid_affected_rows() to find out how many rows were affected by a DELETE, INSERT, REPLACE, or UPDATE statement.

Syntax

```
resource cubrid_query (string $query [, resource $conn_identifier ])
```

- *query* : A SQL query
- *conn_identifier* : The CUBRID connection. If the connection identifier is not specified, the last link opened is assumed.

Return Value

- Success : Request identifier
- Failure or user does not have permission to access the table(s) referenced by the query : FALSE

Example

```
<?php
// This could be supplied by a user, for example
$firstname = 'fred';
$lastname = 'fox';

$conn = cubrid_connect('localhost', 33000, 'foo');

// Formulate Query
// This is the best way to perform an SQL query
// For more examples, see cubrid_real_escape_string()
$query = sprintf("SELECT firstname, lastname, address, age FROM friends WHERE
firstname='%s' AND lastname='%s'",
cubrid_real_escape_string($firstname),
cubrid_real_escape_string($lastname));

// Perform Query
$result = cubrid_query($query);

// Check result
// This shows the actual query sent to CUBRID, and the error. Useful for debugging.
if (!$result) {
    $message = 'Invalid query: ' . cubrid_error() . "\n";
    $message .= 'Whole query: ' . $query;
die($message);

} // Use result
// Attempting to print $result won't allow access to information in the resource
// One of the cubrid result functions must be used
// See also cubrid_result(), cubrid_fetch_array(), cubrid_fetch_row(), etc.
while ($row = cubrid_fetch_assoc($result)) {
echo $row['firstname'];
echo $row['lastname'];
echo $row['address'];
echo $row['age'];
}

// Free the resources associated with the result set
// This is done automatically at the end of the script
cubrid_free_result($result);
?>
```

See Also

- cubrid_unbuffered_query

cubrid_real_escape_string

Description

The **cubrid_real_escape_string**() function returns the escaped string version of the given string. Follow two escape sequence methods should be supported. On the quoted escape sequence, a string quoted with " or ' can be applied when system parameter *ansi_quotes* is set to "yes". If this option is set to "no", only a string quoted with ' can be applied. The default value is "no".

Quoted escape sequence:

- A ' inside a string quoted with ' may be written as ' '
- A " inside a string quoted with " may be written as "" (applied when ansi_quotes=yes)
- A ' inside a string quoted with " needs no special treatment and need not be doubled or escaped. (applied when ansi_quotes=yes)
- In the same way, " inside a string quoted with ' needs no special treatment.

Backslash escape sequence: This sequence is on by system parameter no_backslash_escapes.

The following characters can be escaped by backslash: \', \", \n, \r, \t, \\, \% _.

If this option is set to "no", backslash escaping will work. The default value is "yes".

Syntax

```
string cubrid_real_escape_string (string $unescaped_string [, resource $link_identifier ] )
```

- *unescaped_string* : The string that is to be escaped.
- *link_identifier* : The CUBRID connection. If the link identifier is not specified, the last link opened by cubrid_connect() is assumed.

Return Value

- Success : Escaped characters
- Failure : FALSE

Example

```
< ?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$unescaped_str = ' !"#$%&\'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~';
$escaped_str = cubrid_real_escape_string($unescaped_str);

$len = strlen($unescaped_str);

@cubrid_execute($conn, "DROP TABLE cubrid_test");
cubrid_execute($conn, "CREATE TABLE cubrid_test (t char($len))");
cubrid_execute($conn, "INSERT INTO cubrid_test (t) VALUES('$escaped_str')");

$req = cubrid_execute($conn, "SELECT * FROM cubrid_test");
$row = cubrid_fetch_assoc($req);

var_dump($row);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(1) {
    ["t"]=>
    string(95) " !"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~"
}
```

cubrid_result

Description

The **cubrid_result**() function retrieves the contents of one cell from a CUBRID result set on success, or **FALSE** on failure.

When working on large result sets, you should consider using one of the functions that fetch an entire row. As these functions return the contents of multiple cells in one function call, they're MUCH quicker than **cubrid_result**(). Also, note that specifying a numeric offset for the field argument is much quicker than specifying a fieldname or tablename.fieldname argument.

Syntax

```
string cubrid_result ( resource $result , int $row [, mixed $field= 0 ] )
```

- *result* : Result that comes from a call to cubrid_execute()

- *row* : The row number from the result that's being retrieved. Row numbers start at 0.
- *field* : The name or offset of the field being retrieved. It can be the field's offset, the field's name, or the field's table dot field name (tablename.fieldname). If the column name has been aliased ('select foo as bar from...'), use the alias instead of the column name. If undefined, the first field is retrieved.

Return Value

- Success : Value of a specific field (NULL if value if null)
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$req = cubrid_execute($conn, "SELECT * FROM code");

$result = cubrid_result($req, 0);
var_dump($result);

$result = cubrid_result($req, 0, 1);
var_dump($result);

$result = cubrid_result($req, 5, "f_name");
var_dump($result);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

string(1) "X"
string(5) "Mixed"
string(4) "Gold"
```

cubrid_rollback

Description

The **cubrid_rollback**() function is used to roll back the transaction being executed in the connection referred by the *conn_identifier*. The connection with the server is terminated after the **cubrid_rollback** method is called, but the connection identifier remains valid.

Syntax

```
bool cubrid_rollback (resource $conn_identifier)
```

- *conn_identifier* : Connection identifier

Example

```
<?php
$conn = cubrid_connect("127.0.0.1", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE publishers");

$sql = <<EOD
CREATE TABLE publishers(
pub_id CHAR(3),
pub_name VARCHAR(20),
city VARCHAR(15),
state CHAR(2),
country VARCHAR(15)
)
EOD;

if (!cubrid_execute($conn, $sql)) {
```

```
    printf("Error facility: %d\nError code: %d\nError msg: %s\n",
cubrid_error_code_facility(), cubrid_error_code(), cubrid_error_msg());

    cubrid_disconnect($conn);
    exit;
}

$req = cubrid_prepare($conn, "INSERT INTO publishers VALUES(?, ?, ?, ?, ?)");

$id_list = array("P01", "P02", "P03", "P04");
$name_list = array("Abatis Publishers", "Core Dump Books", "Schadenfreude Press",
"Tenterhooks Press");
$city_list = array("New York", "San Francisco", "Hamburg", "Berkeley");
$state_list = array("NY", "CA", NULL, "CA");
$country_list = array("USA", "USA", "Germany", "USA");

for ($i = 0, $size = count($id_list); $i < $size; $i++) {
    cubrid_bind($req, 1, $id_list[$i]);
    cubrid_bind($req, 2, $name_list[$i]);
    cubrid_bind($req, 3, $city_list[$i]);
    cubrid_bind($req, 4, $state_list[$i]);
    cubrid_bind($req, 5, $country_list[$i]);

    if (!($ret = cubrid_execute($req))) {
        break;
    }
}

if (!$ret) {
    cubrid_rollback($conn);
} else {
    cubrid_commit($conn);

    $req = cubrid_execute($conn, "SELECT * FROM publishers");
    while ($result = cubrid_fetch_assoc($req)) {
    printf("%-3s %-20s %-15s %-3s %-15s\n",
        $result["pub_id"], $result["pub_name"], $result["city"], $result["state"],
$result["country"]);
    }
}

cubrid_disconnect($conn);
?>

The above example will output:

P01     Abatis Publishers              New York             NY     USA
P02     Core Dump Books                San Francisco        CA     USA
P03     Schadenfreude Press            Hamburg                     Germany
P04     Tenterhooks Press              Berkeley             CA     USA
```

See Also

- cubrid_commit
- cubrid_disconnect

cubrid_schema

Description

The **cubrid_schema**() function is used to get specific schema information of a database. You should specify *class_name* to get information related to a specific class, and *attr_name* to get information related to a specific attribute (currently, only used with **CUBRID_SCH_ATTR_PRIVILEGE**).

Syntax

```
array cubrid_schema (resource $conn_identifier, int $schema_type[, string $class_name[,
string $attr_name]])
```

- *conn_identifier* : Connection identifier

- *schema_type* : Type of schema you want to get
- *class_name* : Class from which schema is to be obtained
- *attr_name* : Attribute from which schema is to be obtained

Return Value

- Success : Array in which schema information is contained
- Failure : FALSE

The result of the cubrid_schema() function is returned as a two-dimensional array(column (associative array) * row (numeric array)). The following table shows types of schema and the column structure of the result array to be returned based on the schema type.

Schema	**Column Number**	**Column Name**	**Value**
CUBRID_SCH_CLASS	1	NAME	0 : System class 1 : vclass 2 : class
	2	TYPE	
CUBRID_SCH_VCLASS	1	NAME	1 : vclass
	2	TYPE	
CUBRID_SCH_QUERY_SPEC	1	QUERY_SPEC	
CUBRID_SCH_ATTRIBUTE	1	ATTR_NAME	
	2	DOMAIN	
	3	SCALE	
	4	PRECISION	
	5	INDEXED	1 : indexed
	6	NON NULL	1 : non null
	7	SHARED	1 : shared
	8	UNIQUE	1 : unique
	9	DEFAULT	
	10	ATTR_ORDER	1 : base
	11	CLASS_NAME	
	12	SOURCE_CLASS	
CUBRID_SCH_CLASS_ATTRIBUTE	1	ATTR_NAME	
	2	DOMAIN	
	3	SCALE	
	4	PRECISION	
	5	INDEXED	1 : indexed
	6	NON NULL	1 : non null
	7	SHARED	1 : shared
	8	UNIQUE	1 : unique
	9	DEFAULT	
	10	ATTR_ORDER	1 : base
	11	CLASS_NAME	

	12	SOURCE_CLASS	
CUBRID_SCH_METHOD	1	NAME	
	2	RET_DOMAIN	
	3	ARG_DOMAIN	
CUBRID_SCH_METHOD_FILE	1	METHOD_FILE	
CUBRID_SCH_SUPERCLASS	1	CLASS_NAME	
	2	TYPE	
CUBRID_SCH_SUBCLASS	1	CLASS_NAME	
	2	TYPE	
CUBRID_SCH_CONSTRAINT	1	TYPE	0 : unique 1 : index
	2	NAME	
	3	ATTR_NAME	
CUBRID_SCH_TRIGGER	1	NAME	
	2	STATUS	
	3	EVENT	
	4	TARGET_CLASS	
	5	TARGET_ATTR	
	6	ACTION_TIME	
	7	ACTION	
	8	PRIORITY	
	9	CONDITION_TIME	
	10	CONDITION	
CUBRID_SCH_CLASS_PRIVILEGE	1	CLASS_NAME	
	2	PREVILEGE	
	3	GRANTABLE	
CUBRID_SCH_ATTR_PRIVILEGE	1	ATT_NAME	
	2	PREVILEGE	
	3	GRANTABLE	
CUBRID_SCH_PRIMARY_KEY	1	ATTR_NAME	
	2	KEY_SEQ	1 : base
	3	KEY_NAME	
	4	KEY_NAME	
CUBRID_SCH_IMPORTED_KEYS	1	PKTABLE_NAME	
	2	PKCOLUMN_NAME	
	3	FKTABLE_NAME	
	4	FKCOLUMN_NAME	
	5	KEY_SEQ	
	6	UPDATE_ACTION	0 : cascade 1 : restrict 2 : no action

			3 : set null
	7	DELETE_ACTION	0 : cascade 1 : restrict 2 : no action 3 : set null
	8	FK_NAME	
	9	PK_NAME	
CUBRID_SCH_EXPORTED_KEYS	1	PKTABLE_NAME	
	2	PKCOLUMN_NAME	
	3	FKTABLE_NAME	
	4	FKCOLUMLN_NAME	
	5	KEY_SEQ	
	6	UPDATE_ACTION	0 : cascade 1 : restrict 2 : no action 3 : set null
	7	DELETE_ACTION	0 : cascade 1 : restrict 2 : no action 3 : set null
	8	FK_NAME	
	9	PK_NAME	
CUBRID_SCH_CROSS_REFERENCE	1	PKTABLE_NAME	
	2	PKCOLUMN_NAME	
	3	FKTABLE_NAME	
	4	FKCOLUMLN_NAME	
	5	KEY_SEQ	
	6	UPDATE_ACTION	0 : cascade 1 : restrict 2 : no action 3 : set null
	7	DELETE_ACTION	0 : cascade 1 : restrict 2 : no action 3 : set null
	8	FK_NAME	
	9	PK_NAME	

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

printf("\n--- Primary Key ---\n");
$pk = cubrid_schema($conn, CUBRID_SCH_PRIMARY_KEY, "game");
var_dump($pk);

printf("\n--- Foreign Keys ---\n");
$fk = cubrid_schema($conn, CUBRID_SCH_IMPORTED_KEYS, "game");
var_dump($fk);
```

```
printf("\n--- Column Attribute ---\n");
$attr = cubrid_schema($conn, CUBRID_SCH_ATTRIBUTE, "stadium", "area");
var_dump($attr);

cubrid_disconnect($conn);
?>
The above example will output:

--- Primary Key ---
array(3) {
    [0]=>
    array(4) {
    ["CLASS_NAME"]=>
    string(4) "game"
    ["ATTR_NAME"]=>
    string(12) "athlete_code"
    ["KEY_SEQ"]=>
    string(1) "3"
    ["KEY_NAME"]=>
    string(41) "pk_game_host_year_event_code_athlete_code"
    }
    [1]=>
    array(4) {
    ["CLASS_NAME"]=>
    string(4) "game"
    ["ATTR_NAME"]=>
    string(10) "event_code"
    ["KEY_SEQ"]=>
    string(1) "2"
    ["KEY_NAME"]=>
    string(41) "pk_game_host_year_event_code_athlete_code"
    }
    [2]=>
    array(4) {
    ["CLASS_NAME"]=>
    string(4) "game"
    ["ATTR_NAME"]=>
    string(9) "host_year"
    ["KEY_SEQ"]=>
    string(1) "1"
    ["KEY_NAME"]=>
    string(41) "pk_game_host_year_event_code_athlete_code"
    }
}

--- Foreign Keys ---
array(2) {
    [0]=>
    array(9) {
    ["PKTABLE_NAME"]=>
    string(7) "athlete"
    ["PKCOLUMN_NAME"]=>
    string(4) "code"
    ["FKTABLE_NAME"]=>
    string(4) "game"
    ["FKCOLUMN_NAME"]=>
    string(12) "athlete_code"
    ["KEY_SEQ"]=>
    string(1) "1"
    ["UPDATE_RULE"]=>
    string(1) "1"
    ["DELETE_RULE"]=>
    string(1) "1"
    ["FK_NAME"]=>
    string(20) "fk_game_athlete_code"
    ["PK_NAME"]=>
    string(15) "pk_athlete_code"
    }
    [1]=>
    array(9) {
    ["PKTABLE_NAME"]=>
    string(5) "event"
```

```
    ["PKCOLUMN_NAME"]=>
    string(4) "code"
    ["FKTABLE_NAME"]=>
    string(4) "game"
    ["FKCOLUMN_NAME"]=>
    string(10) "event_code"
    ["KEY_SEQ"]=>
    string(1) "1"
    ["UPDATE_RULE"]=>
    string(1) "1"
    ["DELETE_RULE"]=>
    string(1) "1"
    ["FK_NAME"]=>
    string(18) "fk_game_event_code"
    ["PK_NAME"]=>
    string(13) "pk_event_code"
    }
}

--- Column Attribute ---
array(1) {
    [0]=>
    array(13) {
    ["ATTR_NAME"]=>
    string(4) "area"
    ["DOMAIN"]=>
    string(1) "7"
    ["SCALE"]=>
    string(1) "2"
    ["PRECISION"]=>
    string(2) "10"
    ["INDEXED"]=>
    string(1) "0"
    ["NON_NULL"]=>
    string(1) "0"
    ["SHARED"]=>
    string(1) "0"
    ["UNIQUE"]=>
    string(1) "0"
    ["DEFAULT"]=>
    NULL ["ATTR_ORDER"]=>
    string(1) "4"
    ["CLASS_NAME"]=>
    string(7) "stadium"
    ["SOURCE_CLASS"]=>
    string(7) "stadium"
    ["IS_KEY"]=>
    string(1) "0"
    }
}
```

cubrid_seq_drop

Description

The **cubrid_seq_drop**() function is used to drop elements from the given **SEQUENCE** type attribute in the database.

Syntax

```
bool cubrid_seq_drop(resource $conn_identifier, string $oid, string $attr_name, int $index)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance
- *index* : Index of the element to be dropped. The default value is 1.

Return Value

- Success : TRUE

- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c sequence(int),
d char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_col_get($conn, $oid, "c");
var_dump($attr);

cubrid_seq_drop($conn, $oid, "c", 4);

$attr = cubrid_col_get($conn, $oid, "c");
var_dump($attr);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(4) {
    [0]=>
    string(2) "11"
    [1]=>
    string(2) "22"
    [2]=>
    string(2) "33"
    [3]=> string(3) "333"
    }
array(3) {
    [0]=>
    string(2) "11"
    [1]=>
    string(2) "22"
    [2]=>
    string(2) "33"
}
```

See Also

- cubrid_seq_insert
- cubrid_seq_put

cubrid_seq_insert

Description

The **cubrid_seq_inset**() function is used to insert an element into a specific position of a SEQUENCE type attribute.

Syntax

```
bool cubrid_seq_insert (resource $conn_identifier, string $oid, string $attr_name, int
$index, string $seq_element)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance

- *index* : Position into which the new element is to be inserted (default value: 1)
- *seq_string* : Content of the element to be inserted

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c sequence(int),
d char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_col_get($conn, $oid, "c");
var_dump($attr);

cubrid_seq_insert($conn, $oid, "c", 5, "44");

$attr = cubrid_col_get($conn, $oid, "c");
var_dump($attr);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(4) {
    [0]=>
    string(2) "11"
    [1]=>
    string(2) "22"
    [2]=>
    string(2) "33"
    [3]=> string(3) "333"
}
array(5) {
    [0]=>
    string(2) "11"
    [1]=>
    string(2) "22"
    [2]=>
    string(2) "33"
    [3]=>
    string(3) "333"
    [4]=>
    string(2) "44"
}
```

See Also

- cubrid_seq_drop
- cubrid_seq_put

cubrid_seq_put

Description

The **cubrid_seq_put**() function is used to change the content of an element of the given SEQUENCE type attribute.

Syntax

```
bool cubrid_seq_put (resource $conn_identifier, string $oid, string $attr_name, int index,
string $seq_element)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance
- *index* : Index of the element to be changed (default value: 1)
- *seq_element* : Content of the element to be changed

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c sequence(int),
d char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_col_get($conn, $oid, "c");
var_dump($attr);

cubrid_seq_put($conn, $oid, "c", 1, "111");

$attr = cubrid_col_get($conn, $oid, "c");
var_dump($attr);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(4) {
    [0]=>
    string(2) "11"
    [1]=>
    string(2) "22"
    [2]=>
    string(2) "33"
    [3]=>
    string(3) "333"
}
array(4) {
    [0]=>
    string(3) "111"
    [1]=>
    string(2) "22"
    [2]=>
    string(2) "33"
```

```
    [3]=> string(3)
    "333"
}
```

See Also

- cubrid_seq_insert
- cubrid_seq_drop

cubrid_set_add

Description

The **cubrid_set_add**() function is used to insert an element to the given SET type (set, multiset) attribute.

Syntax

```
bool cubrid_set_add (resource $conn_identifier, string $oid, string $attr_name, string
$set_element)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance
- *set_string* : Content of the element to be inserted

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_col_get($conn, $oid, "b");
var_dump($attr);

cubrid_set_add($conn, $oid, "b", "4");

$attr = cubrid_col_get($conn, $oid, "b");
var_dump($attr);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(3) {
  [0]=>
  string(1) "1"
  [1]=>
  string(1) "2"
  [2]=>
  string(1) "3"
}
```

```
array(4) {
  [0]=>
  string(1) "1"
  [1]=>
  string(1) "2"
  [2]=>
  string(1) "3"
  [3]=>
  string(1) "4"
}
```

See Also

- cubrid_set_drop

cubrid_set_autocommit

Description

The **cubrid_set_autocommit**() function is used to set the status of CUBRID database connection auto-commit mode of the current database connection. This function just turns on/off the auto-commit mode. When this function is called, concurrent transactions are committed regardless of the auto-commit mode.

Syntax

```
bool cubrid_set_autocommit(resource $conn_identifier, int $mode)
```

- *conn_identifier* : Connection identifier
- mode : Whether to turn on auto-commit or not. It should be CUBRID_AUTOCOMMIT_FALSE or CUBRID_AUTOCOMMIT_TRUE.

Return Value

- Success : TRUE
- Failure : FALSE

See Also

- cubrid_get_autocommit

cubrid_set_db_parameter

Description

The **cubrid_set_db_parameter**() function is used to set the CUBRID system parameters. It can set the following CUBRID system parameters:

- **CUBRID_PARAM_ISOLATION_LEVEL** : Transaction isolation level. For more information, see SET TRANSACTION ISOLATION LEVEL.
- **CUBRID_PARAM_LOCK_TIMEOUT** : Lock timeout. Time when transaction lock is held.

Syntax

```
bool cubrid_set_db_parameter ( resource
$conn_identifier, int $param_type, int $param_value)
```

- *conn_identifier* : Connection identifier
- *param_type* : System parameter type
- *param_value* : System parameter value

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
$conn = cubrid_connect("localhost", 33000, "demodb");

$params = cubrid_get_db_parameter($conn);
var_dump($params);

cubrid_set_autocommit($conn, CUBRID_AUTOCOMMIT_TRUE);
cubrid_set_db_parameter($conn, CUBRID_PARAM_ISOLATION_LEVEL, 2);

$params_new = cubrid_get_db_parameter($conn);
var_dump($params_new);

cubrid_disconnect($conn);
?>

The above example will output:

array(4) {
    ["PARAM_ISOLATION_LEVEL"]=>
    int(3)
    ["PARAM_LOCK_TIMEOUT"]=>
    int(-1)
    ["PARAM_MAX_STRING_LENGTH"]=>
    int(1073741823)
    ["PARAM_AUTO_COMMIT"]=>
    int(0)
}
array(4) {
    ["PARAM_ISOLATION_LEVEL"]=>
    int(2)
    ["PARAM_LOCK_TIMEOUT"]=>
    int(-1)
    ["PARAM_MAX_STRING_LENGTH"]=>
    int(1073741823)
    ["PARAM_AUTO_COMMIT"]=>
    int(1)
}
```

See Also

- cubrid_get_db_parameter

cubrid_set_drop

Description

The cubrid_set_drop() function is used to drop an element from the given SET type (set, multiset) attribute.

Syntax

```
bool cubrid_set_drop (resource $conn_identifier, string $oid, string $attr_name, string
$set_element)
```

- *conn_identifier* : Connection identifier
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance
- *set_element* : Content of the element to be dropped.

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
<?php
```

```
$conn = cubrid_connect("localhost", 33000, "demodb");

@cubrid_execute($conn, "DROP TABLE foo");
cubrid_execute($conn, "CREATE TABLE foo(a int AUTO_INCREMENT, b set(int), c list(int), d
char(10))");
cubrid_execute($conn, "INSERT INTO foo(a, b, c, d) VALUES(1, {1,2,3}, {11,22,33,333},
'a')");

$req = cubrid_execute($conn, "SELECT * FROM foo", CUBRID_INCLUDE_OID);

cubrid_move_cursor($req, 1, CUBRID_CURSOR_FIRST);
$oid = cubrid_current_oid($req);

$attr = cubrid_col_get($conn, $oid, "b");
var_dump($attr);

cubrid_set_drop($conn, $oid, "b", "1");

$attr = cubrid_col_get($conn, $oid, "b");
var_dump($attr);

cubrid_close_request($req);
cubrid_disconnect($conn);
?>

The above example will output:

array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "3"
}
array(2) {
    [0]=>
    string(1) "2"
    [1]=>
    string(1) "3"
}
```

See Also

- cubrid_set_add

cubrid_unbuffered_query

Description

The **cubrid_unbuffered_query**() function sends a SQL query (multiple queries are not supported) to CUBRID, without fetching and buffering the result rows automatically, as cubrid_execute() does. On the one hand, this saves a considerable amount of memory with SQL queries that produce large result sets. On the other hand, you can start working on the result set immediately after the first row has been retrieved: you don't have to wait until the complete SQL query has been performed. When using multiple DB-connects, you have to specify the optional parameter *link_identifier*.

Syntax

```
resource cubrid_unbuffered_query ( string $query [, int $conn_identifier ] )
```

- *query* : A SQL query
- *link_identifier* : The CUBRID connection. If the connection identifier is not specified, the last link opened by cubrid_connect() is assumed.

Note The benefits of **cubrid_unbuffered_query**() come at a cost: you cannot use cubrid_num_rows() and cubrid_data_seek() on a result set returned from **cubrid_unbuffered_query**().

Return Value

- Returns **TRUE** on success, or **FALSE** on error.
- For other type of SQL statements such as **UPDATE**, **DELETE**, **DROP**, etc, **cubrid_unbuffered_query**() returns **TRUE** on success or **FALSE** on error.

Example

```
< ?php
$result = cubrid_unbuffered_query("INSERT INTO mytable (product) values ('kossu')", $link)
if (!$result) {
echo 'Could not run query: ' .cubrid_error_msg()
exit
}
printf("Last inserted record has id %d\n", cubrid_insert_id())
?>
```

cubrid_version

Description

The **cubrid_version**() function is used to check the version information of the CUBRID PHP module.

Syntax

```
string cubrid_version ()
```

Return Value

- n version information (e.g. "1.2.0")

Example

```
<?php
printf("%-30s %s\n", "CUBRID PHP Version:", cubrid_version());

printf("\n");

$conn = cubrid_connect("localhost", 33088, "demodb");

if (!$conn) {
    die('Connect Error ('. cubrid_error_code() .')' . cubrid_error_msg());
}

$db_params = cubrid_get_db_parameter($conn);

while (list($param_name, $param_value) = each($db_params)) {
    printf("%-30s %s\n", $param_name, $param_value);
}

printf("\n");

$server_info = cubrid_get_server_info($conn);
$client_info = cubrid_get_client_info();

printf("%-30s %s\n", "Server Info:", $server_info);
printf("%-30s %s\n", "Client Info:", $client_info);

printf("\n");

$charset = cubrid_get_charset($conn);

printf("%-30s %s\n", "CUBRID Charset:", $charset);
cubrid_disconnect($conn);
?>

The above example will output:

CUBRID PHP Version:             8.3.1.0005
```

```
PARAM_ISOLATION_LEVEL          3
LOCK_TIMEOUT                   -1
MAX_STRING_LENGTH              1073741823
PARAM_AUTO_COMMIT                0

Server Info:                   8.3.1.0173
Client Info:                   8.3.1

CUBRID Charset:                iso8859-1
```

See Also

- cubrid_error_code
- cubrid_error_code_facility

CCI API

CCI Overview

Overview

The CCI (C Client Interface) is an interface that exists between the CUBRID broker and the application client, through which a C-based application client can access the CUBRID database server using a broker. This interface is also used as an infrastructure for making tools that utilize CAS (e.g. PHP and ODBC). The CUBRID broker delivers the query received from an application client to the broker, and transfers the execution result to the client.

A header file and library files are required to use CCI.

- Header file : $CUBRID/include/**cas_cci.h**
- Library file :
- $CUBRID/lib/**libcascci.so** (Windows : **cascci.dll**)
- $CUBRID/lib/**libcascci.a** (Windows : **cascci.lib**)

Writing CCI Application Program

The basic steps used for writing programs are as follows, and a step for binding the data to a variable is added to use the prepared statement. The steps are implemented in example codes 1 and 2. Auto-commit mode is supported only for SELECT statements when the SELECT_AUTO_COMMIT parameter has been set to ON. Otherwise, you must explicitly commit or roll back the transaction by using the cci_end_tran() function. For details about how to configure and use auto-commit mode in CCI, see [Performance Tuning > Broker Configuration > Parameter By Broker > SELECT_AUTO_COMMIT] and cci_end_tran().

- Opening a database connection handle (related function : cci_connect(), cci_connect_with_url())
- Preparing an SQL statement (related function : cci_prepare())
- Binding data to a prepared SQL statement (related function : cci_bind_param())
- Executing a prepared SQL statement (related function : cci_execute())
- Processing the execution result (related function : cci_cursor(), cci_fetch(), cci_get_data(), cci_get_result_info())
- Closing the request handle (related function : cci_close_req_handle())
- Closing a database connection handle (related function : cci_disconnect())

Example 1

```
//Example to execute a simple query
#include <stdio.h>
#include "cas_cci.h"
#define BUFSIZE  (1024)

int
main (void)
{
  int con = 0, req = 0, col_count = 0, i, ind;
  int error;
  char *data;
  T_CCI_ERROR cci_error;
  T_CCI_COL_INFO *col_info;
  T_CCI_SQLX_CMD cmd_type;
  char *query = "select * from code";

//getting a connection handle for a connection with a server
  con = cci_connect ("localhost", 44000, "demodb", "dba", "");
  if (con < 0)
    {
      printf ("cannot connect to database\n");
      return 1;
    }
```

```
//preparing the SQL statement
  req = cci_prepare (con, query, 0, &cci_error);
  if (req < 0)
    {
      printf ("prepare error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }

//getting column information when the prepared statement is the SELECT query
  col_info = cci_get_result_info (req, &cmd_type, &col_count);
  if (col_info == NULL)
    {
      printf ("get_result_info error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }

//Executing the prepared SQL statement
  error = cci_execute (req, 0, 0, &cci_error);
  if (error < 0)
    {
      printf ("execute error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }
  while (1)
    {

//Moving the cursor to access a specific tuple of results
      error = cci_cursor (req, 1, CCI_CURSOR_CURRENT, &cci_error);
      if (error == CCI_ER_NO_MORE_DATA)
        {
          break;
        }
      if (error < 0)
        {
          printf ("cursor error: %d, %s\n", cci_error.err_code,
                  cci_error.err_msg);
          goto handle_error;
        }

//Fetching the query result into a client buffer
      error = cci_fetch (req, &cci_error);
      if (error < 0)
        {
          printf ("fetch error: %d, %s\n", cci_error.err_code,
                  cci_error.err_msg);
          goto handle_error;
        }
      for (i = 1; i <= col_count; i++)
        {

//Getting data from the fetched result
          error = cci_get_data (req, i, CCI_A_TYPE_STR, &data, &ind);
          if (error < 0)
            {
              printf ("get_data error: %d, %d\n", error, i);
              goto handle_error;
            }
          printf ("%s\t|", data);
        }
      printf ("\n");
    }

//Closing the request handle
  error = cci_close_req_handle (req);
  if (error < 0)
    {
      printf ("close_req_handle error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }
```

```
//Disconnecting with the server
  error = cci_disconnect (con, &cci_error);
  if (error < 0)
    {
      printf ("error: %d, %s\n", cci_error.err_code, cci_error.err_msg);
      goto handle_error;
    }

  return 0;

handle_error:
  if (req > 0)
    cci_close_req_handle (req);
  if (con > 0)
    cci_disconnect (con, &cci_error);

  return 1;
}
```

Example 2

```
//Example to execute a query with a bind variable

char *query = "select * from nation where name = ?";
  char namebuf[128];

//getting a connection handle for a connection with a server
  con = cci_connect ("localhost", 44000, "demodb", "dba", "");
  if (con < 0)
    {
      printf ("cannot connect to database ");
      return 1;
    }

//preparing the SQL statement
  req = cci_prepare (con, query, 0, &cci_error);
  if (req < 0)
    {
      printf ("prepare error: %d, %s ", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }

//Binding date into a value
  strcpy (namebuf, "Korea");
  error =
    cci_bind_param (req, 1, CCI_A_TYPE_STR, &namebuf, CCI_U_TYPE_STRING,
                    CCI_BIND_PTR);
  if (error < 0)
    {
      printf ("bind_param error: %d ", error);
      goto handle_error;
    }
```

Using BLOB/CLOB with CCI

Storing LOB Data

You can create LOB data file and bind the data by using the following functions in CCI applications.

- Creating LOB data file (related function : cci_blob_new( ), cci_blob_write( ))
- Binding LOB data (related function : cci_bind_param( ))
- Freeing memory of LOB structure (related function : cci_blob_free( ))

Example 1

```
int con = 0; /* connection handle */
int req = 0; /* request handle */
int res;
```

```
int n_executed;
int i;
T_CCI_ERROR error;
T_CCI_BLOB blob = NULL;
char data[1024] = "bulabula";

con = cci_connect ("localhost", 33000, "tdb", "PUBLIC", "");
if (con < 0) {
goto handle_error;
}
req = cci_prepare (con, "insert into doc (doc_id, content) values (?,?)", 0, &error);
if (req< 0)
{
goto handle_error;
}

res = cci_bind_param (req, 1 /* binding index*/, CCI_A_TYPE_STR, "doc-10", &ind,
CCI_U_TYPE_STRING);

/* Creating an empty LOB data file
res = cci_blob_new (con, &blob, &error);
res = cci_blob_write (con, blob, 0 /* start position */, 1024 /* length */, data, &error);

/* Binding BLOB data */
res = cci_bind_param (req, 2 /* binding index*/, CCI_A_TYPE_BLOB, (void *)blob,
CCI_U_TYPE_BLOB, CCI_BIND_PTR);

n_executed = cci_execute (req, 0, 0, &error);
if (n_executed < 0)
{
goto handle_error
}

/* Memory free */
cci_blob_free(blob);
return 0;

handle_error:
if (blob != NULL)
{
cci_blob_free(blob);
}
if (req > 0)
{
cci_close_req_handle (req);
}
if (con > 0)
{
cci_disconnect(con, &error);
}
return -1;
```

Getting LOB Data

You can get LOB data by using the following functions in CCI applications. Note that if you enter data in LOB type colulmn, the actual LOB data is stored externally and Locator value referring to the file is stored in LOB type column itself. Therefore, you must call the cci_blob_read() function (not the cci_get_data() function) to get LOB data stored in the file.

- Getting LOB type column value (Locator) (related function : cci_get_data( ))
- Getting LOB data (related function : cci_blob_read( ))
- Freeing memory of LOB structure (related function : cci_blob_free( ))

Example

```
int con = 0; /* connection handle */
int req = 0; /* request handle */
int ind; /* NULL indicator, 0 if not NULL, -1 if NULL*/
int res;
int i;
```

```
T_CCI_ERROR error;
T_CCI_BLOB blob;
char buffer[1024];

con = cci_connect ("localhost", 33000, "image_db", "PUBLIC", "");
if (con < 0)
 {
  goto handle_error;
 }
req = cci_prepare (con, "select content from doc_t", 0 /*flag*/, &error);
if (req< 0)
 {
  goto handle_error;
 }
res = cci_execute (req, 0/*flag*/, 0/*max_col_size*/, &error);
res = cci_fetch_size (req, 100 /* fetch size */);

while (1) {
  res = cci_cursor (req, 1/* offset */, CCI_CURSOR_CURRENT/* cursor position */, &error);
  if (res == CCI_ER_NO_MORE_DATA)
      {
      break;
      }
  res = cci_fetch (req, &error);

/* Fetching CLOB Locator */
  res = cci_get_data (req, 1 /* colume index */, CCI_A_TYPE_BLOB,
        (void *)&blob /* BLOB handle */, &ind /* NULL indicator */);

/* Fetching CLOB data */
res = cci_blob_read (con, blob, 0 /* start position */, 1024 /* length */, buffer, &error);
printf ("content = %s\n", buffer);
}
/* Memory free */
cci_blob_free(blob);
res=cci_close_req_handle(req);
res = cci_disconnect (con, &error);
return 0;

handle_error:
if (req > 0)
     {
      cci_close_req_handle (req);
     }
if (con > 0)
    {
     cci_disconnect(con, &error);
    }
return -1;
```

CCI Error Code and Message

The following table shows the error codes and their messages of CCI.

Error Code	Error Message	Note
CCI_ER_ALLOC_CON_HANDLE	"Cannot allocate connection handle"	
CCI_ER_ATYPE	"Invalid T_CCI_A_TYPE value"	
CCI_ER_BIND_ARRAY_SIZE	"Array binding size is not specified"	
CCI_ER_BIND_INDEX	"Parameter index is out of range"	Index that binds data is not valid.
CCI_ER_COLUMN_INDEX	"Column index is out of range"	
CCI_ER_COMMUNICATION	"Cannot communicate with	

	server"	
CCI_ER_CON_HANDLE	"Invalid connection handle"	
CCI_ER_CONNECT	"Cannot connect to CUBRID CAS"	Fails to connect the CAS when trying connection to the server.
CCI_ER_DELETED_TUPLE	"Current row was deleted"	
CCI_ER_FILE	"Cannot open file"	Fails to open/read/write a file.
CCI_ER_HOSTNAME	"Unknown host name"	
CCI_ER_INVALID_CURSOR_POS	"Invalid cursor position"	
CCI_ER_INVALID_URL	"Invalid url string"	
CCI_ER_ISOLATION_LEVEL	"Unknown transaction isolation level"	
CCI_ER_NO_MORE_DATA	"Invalid cursor position"	
CCI_ER_NO_MORE_MEMORY	"Memory allocation error"	Insufficient memory
CCI_ER_OBJECT	"Invalid oid string"	
CCI_ER_OID_CMD	"Invalid T_CCI_OID_CMD value"	
CCI_ER_TRAN_TYPE	"Unknown transaction type"	
CCI_ER_PARAM_NAME	"Invalid T_CCI_DB_PARAM value"	
CCI_ER_REQ_HANDLE	"Cannot allocate request handle"	
CCI_ER_SAVEPOINT_CMD	"Invalid T_CCI_SAVEPOINT_CMD value"	Invalid T_CCI_SAVEPOINT_CMD value is used as an argument of cci_savepoint() function.
CCI_ER_SET_INDEX	"Invalid set index"	Invalid index is specified when an set element in the T_SET is retrieved.
CCI_ER_STRING_PARAM	"Invalid string argument"	string parameter is NULL or an empty string.
CCI_ER_THREAD_RUNNING	"Thread is running"	The thread is still executed when cci_execute() is executed with CCI_EXEC_THREAD flaged and check the result of thread execution through cci_get_thread_result().
CCI_ER_TRAN_TYPE	"Unknown transaction type"	Connection to the server has succeeded, connection to a database fails.
CCI_ER_TYPE_CONVERSION	"Type conversion error"	Cannot convert the given value into an actual data type.
CCI_ER_DBMS CAS_ER_DBMS	"CUBRID DBMS Error"	Fails to database connection.
CAS_ER_COLLECTION_DOMAIN	"Heterogeneous set is not supported"	Not supported set type.
CAS_ER_COMMUNICATION	"Cannot receive data from	

	client"	
CAS_ER_DB_VALUE	"Cannot make DB_VALUE"	
CAS_ER_DBSERVER_DISCONNECTED	"Cannot communicate with DB Server"	
CAS_ER_FREE_SERVER	"Cannot process the request. Try again later"	Cannot assign CAS.
CAS_ER_INVALID_CALL_STMT	"Illegal CALL statement"	
CAS_ER_NO_MORE_DATA	"Invalid cursor position"	
CAS_ER_NO_MORE_MEMORY	"Memory allocation error"	
CAS_ER_NO_MORE_RESULT_SET	"No More Result"	
CAS_ER_NOT_AUTHORIZED_CLIENT	"Authorization error"	Access is denied.
CAS_ER_NOT_COLLECTION	"The attribute domain must be the set type"	No set type.
CAS_ER_NUM_BIND	"Invalid parameter binding value argument"	The number of data to be bound is not matched with the number of delivered data.
CAS_ER_OBJECT	"Invalid oid"	
CAS_ER_OPEN_FILE	"Cannot open file"	
CAS_ER_PARAM_NAME	"Invalid T_CCI_DB_PARAM value"	Invalid get_db_parameter and , set_db_parameter parameter name.
CAS_ER_QUERY_CANCEL	"Cannot cancel the query"	
CAS_ER_UNKNOWN_U_TYPE	"Invalid T_CCI_U_TYPE value"	
CAS_ER_TYPE_CONVERSION	"Type conversion error"	
CAS_ER_SCHEMA_TYPE	"Invalid T_CCI_SCH_TYPE value"	
CAS_ER_STMT_POOLING	"Invalid plan"	
CAS_ER_TRAN_TYPE	"Invalid transaction type argument"	
CAS_ER_TYPE_CONVERSION	"Type conversion error"	
CAS_ER_UNKNOWN_U_TYPE	"Invalid T_CCI_U_TYPE value"	
CAS_ER_VERSION	"Version mismatch"	Invalid Server and Client version.

C Type Definition

Name	Type	Member	Description
T_CCI_ERROR	struct	char err_msg[1024]	Representation of database error info
		int err_code	
T_CCI_BIT	struct	int size	Representation of bit type
		char *buf	
T_CCI_DATE	struct	short yr	Representation of timestamp, date,
		short mon	

		short day	time type
		short hh	
		short mm	
		short ss	
		short ms	
T_CCI_SET	void*		Representation of set type
T_CCI_COL_INFO	struct	**T_CCI_U_TYPE** type	Representation of column information for the **SELECT** statement
		char is_non_null	
		short scale	
		int precision	
		char *col_name	
		char *real_attr	
		char *class_name	
T_CCI_QUERY_RESULT	struct	int result_count	Results of batch execution
		int stmt_type	
		char *err_msg	
		char oid[32]	
T_CCI_PARAM_INFO	struct	**T_CCI_PARAM_MODE** mode	Representation of input parameter info
		T_CCI_U_TYPE type	
		short scale	
		int precision	
T_CCI_U_TYPE	enum	**CCI_U_TYPE_UNKNOWN**	Database type info
		CCI_U_TYPE_NULL	
		CCI_U_TYPE_CHAR	
		CCI_U_TYPE_STRING	
		CCI_U_TYPE_NCHAR	
		CCI_U_TYPE_VARNCHAR	
		CCI_U_TYPE_BIT	
		CCI_U_TYPE_VARBIT	
		CCI_U_TYPE_NUMERIC	
		CCI_U_TYPE_INT	
		CCI_U_TYPE_SHORT	
		CCI_U_TYPE_MONETARY	
		CCI_U_TYPE_FLOAT	
		CCI_U_TYPE_DOUBLE	
		CCI_U_TYPE_DATE	
		CCI_U_TYPE_TIME	
		CCI_U_TYPE_TIMESTAMP	
		CCI_U_TYPE_SET	

		CCI_U_TYPE_MULTISET	
		CCI_U_TYPE_SEQUENCE	
		CCI_U_TYPE_OBJECT	
		CCI_U_TYPE_BIGINT	
		CCI_U_TYPE_DATETIME	
T_CCI_A_TYPE	enum	**CCI_A_TYPE_STR**	Representation of type info used in API
		CCI_A_TYPE_INT	
		CCI_A_TYPE_FLOAT	
		CCI_A_TYPE_DOUBLE	
		CCI_A_TYPE_BIT	
		CCI_A_TYPE_DATE	
		CCI_A_TYPE_SET	
		CCI_A_TYPE_BIGINT	
		CCI_TYPE_BLOB	
		CCI_TYPE_CLOB	
T_CCI_DB_PARAM	enum	**CCI_PARAM_ISOLATION_LEVEL**	System parameter name
		CCI_PARAM_LOCK_TIMEOUT	
		CCI_PARAM_MAX_STRING_LENGTH	
		CCI_PARAM_AUTO_COMMIT	
T_CCI_SCH_TYPE	enum	**CCI_SCH_CLASS**	
		CCI_SCH_VCLASS	
		CCI_SCH_QUERY_SPEC	
		CCI_SCH_ATTRIBUTE	
		CCI_SCH_CLASS_ATTRIBUTE	
		CCI_SCH_METHOD	
		CCI_SCH_CLASS_METHOD	
		CCI_SCH_METHOD_FILE	
		CCI_SCH_SUPERCLASS	
		CCI_SCH_SUBCLASS	
		CCI_SCH_CONSTRAIT	
		CCI_SCH_TRIGGER	
		CCI_SCH_CLASS_PRIVILEGE	
		CCI_SCH_ATTR_PRIVILEGE	
		CCI_SCH_DIRECT_SUPER_CLASS	
		CCI_SCH_PRIMARY_KEY	
		CCI_SCH_IMPORTED_KEYS	
		CCI_SCH_EXPORTED_KEYS	
		CCI_SCH_CROSS_REFERENCE	
T_CCI_CUBRID_STMT	enum	**CUBRID_STMT_ALTER_CLASS**	
		CUBRID_STMT_ALTER_SERIAL	

	CUBRID_STMT_COMMIT_WORK
	CUBRID_STMT_REGISTER_DATABASE
	CUBRID_STMT_CREATE_CLASS
	CUBRID_STMT_CREATE_INDEX
	CUBRID_STMT_CREATE_TRIGGER
	CUBRID_STMT_CREATE_SERIAL
	CUBRID_STMT_DROP_DATABASE
	CUBRID_STMT_DROP_CLASS
	CUBRID_STMT_DROP_INDEX
	CUBRID_STMT_DROP_LABEL
	CUBRID_STMT_DROP_TRIGGER
	CUBRID_STMT_DROP_SERIAL
	CUBRID_STMT_EVALUATE
	CUBRID_STMT_RENAME_CLASS
	CUBRID_STMT_ROLLBACK_WORK
	CUBRID_STMT_GRANT
	CUBRID_STMT_REVOKE
	CUBRID_STMT_STATISTICS
	CUBRID_STMT_INSERT
	CUBRID_STMT_SELECT
	CUBRID_STMT_UPDATE
	CUBRID_STMT_DELETE
	CUBRID_STMT_CALL
	CUBRID_STMT_GET_ISO_LVL
	CUBRID_STMT_GET_TIMEOUT
	CUBRID_STMT_GET_OPT_LVL
	CUBRID_STMT_SET_OPT_LVL
	CUBRID_STMT_SCOPE
	CUBRID_STMT_GET_TRIGGER
	CUBRID_STMT_SET_TRIGGER
	CUBRID_STMT_SAVEPOINT
	CUBRID_STMT_PREPARE
	CUBRID_STMT_ATTACH
	CUBRID_STMT_USE
	CUBRID_STMT_REMOVE_TRIGGER
	CUBRID_STMT_RENAME_TRIGGER
	CUBRID_STMT_ON_LDB
	CUBRID_STMT_GET_LDB
	CUBRID_STMT_SET_LDB
	CUBRID_STMT_GET_STATS

		CUBRID_STMT_CREATE_USER	
		CUBRID_STMT_DROP_USER	
		CUBRID_STMT_ALTER_USER	
T_CCI_CURSOR_POS	enum	**CCI_CURSOR_FIRST**	
		CCI_CURSOR_CURRENT	
		CCI_CURSOR_LAST	
T_CCI_TRAN_ISOLATION	enum	**TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE**	
		TRAN_COMMIT_CLASS_COMMIT_INSTANCE	
		TRAN_REP_CLASS_UNCOMMIT_INSTANCE	
		TRAN_REP_CLASS_COMMIT_INSTANCE	
		TRAN_REP_CLASS_REP_INSTANCE	
		TRAN_SERIALIZABLE	
T_CCI_PARAM_MODE	enum	**CCI_PARAM_MODE_UNKNOWN**	
		CCI_PARAM_MODE_IN	
		CCI_PARAM_MODE_OUT	
		CCI_PARAM_MODE_INOUT	

Note If a string longer than defined size in a column is inserted(INSERT) or updated(UPDATE), the string will be truncated.

cci_bind_param

Description

This function is used to bind data in the *bind* variable of prepared statement. Converts *value* of the given *a_type* to an actual binding type and saves it. Subsequently, whenever cci_execute() is called, the saved data is sent to the server. If **cci_bind_param**() is called multiple times for the same *index*, the last set value is configured.

If **NULL** is bound to the database, there can be two scenarios.

- *value* is a **NULL** pointer.
- *u_type* is **CCI_U_TYPE_NULL**.

If **CCI_BIND_PTR** is configured for *flag*, the pointer of *value* variable is copied (shallow copy), but no value is copied. If it is not configured for *flag*, the value of *value* variable is copied (deep copy) by allocating memory. If multiple columns are bound by using the same memory buffer, **CCI_BIND_PTR** must not be configured for the *flag*.

T_CCI_A_TYPE is a C language type that is used in CCI application programs, and consists of primitive types such as int and float and user-defined types defined by CCI such as **T_CCI_BIT** and **T_CCI_DATE**. The identitier for each type is defined as shown in the table below.

a_type	value Type
CCI_A_TYPE_STR	char**
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	**T_CCI_BIT***
CCI_A_TYPE_SET	T_CCI_SET

CCI_A_TYPE_DATE	**T_CCI_DATE***
CCI_A_TYPE_BIGINT	int64_t* (For Windows : __int64*)
CCI_A_TYPE_BLOB	T_CCI_BLOC
CCI_A_TYPE_CLOB	T_CCI_CLOB

T_CCI_U_TYPE is a type supported by the CUBRID database. For the definition of the identifier for each type, see the table below. These two types are used in the **cci_bind_param**() function to deliver the information required to convert the A type data that can be understood by the C language to the U type data that can be understood by the database. **T_CCI_A_TYPE** and **T_CCI_U_TYPE** enums are all defined in the **cas_cci.h file**.

u_type	value Type
CCI_U_TYPE_CHAR	char**
CCI_U_TYPE_STRING	char**
CCI_U_TYPE_NCHAR	char**
CCI_U_TYPE_VARNCHAR	char**
CCI_U_TYPE_BIT	T_CCI_BIT*
CCI_U_TYPE_VARBIT	T_CCI_BIT*
CCI_U_TYPE_NUMERIC	char**
CCI_U_TYPE_INT	int*
CCI_U_TYPE_SHORT	int*
CCI_U_TYPE_MONETARY	Double*
CCI_U_TYPE_FLOAT	float*
CCI_U_TYPE_DOUBLE	Double*
CCI_U_TYPE_DATE	T_CCI_DATE*
CCI_U_TYPE_TIME	T_CCI_DATE*
CCI_U_TYPE_TIMESTAMP	T_CCI_DATE*
CCI_U_TYPE_OBJECT	char**
CCI_U_TYPE_BIGINT	int64_t* (Windows : __int64*)
CCI_U_TYPE_DATETIME	T_CCI_DATE*

Syntax

```
int cci_bind_param(int req_handle, int index, T_CCI_A_TYPE a_type, void *value,
T_CCI_U_TYPE u_type, char flag)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *index* : (IN) One-based binding location it starts with 1.
- *a_type* : (IN) Data type of *value*
- *value* : (IN) Data value to be bound
- *u_type* : (IN) Data type to be applied to the database
- *flag* : (IN) bind_flag (**CCI_BIND_PTR**)

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_BIND_INDEX**
- **CCI_ER_ATYPE**
- **CCI_ER_NO_MORE_MEMORY**

cci_bind_param_array

Description

This function is used to bind a parameter array for a prepared req_handle. Subsequently, whenever cci_execute_array() occurs, data is sent to the server by the saved *value* pointer. If **cci_bind_param_array**() is called multiple times for the same *index*, the last configured value is used. If **NULL** is bound to the data, a non-zero value is configured to *null_ind*.

If *value* is a **NULL** pointer, or *u_type* is **CCI_U_TYPE_NULL**, all data are bound to **NULL** and the data buffer used by *value* cannot be reused.

For the data type of *value* for *a_type*, see the cci_bind_param() function description.

Syntax

```
int cci_bind_param_array(int req_handle, int index, T_CCI_A_TYPE a_type, void *value, int *null_ind, T_CCI_U_TYPE u_type)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *index* : (IN) Binding location
- *a_type* : (IN) Data type of *value*
- *value* : (IN) Data value to be bound
- *null_ind* : (IN) **NULL** indicator array (0 : not **NULL**, 1 : **NULL**)
- *u_type* : (IN) Data type to be applied to the database.

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_BIND_INDEX**
- **CCI_ER_ATYPE**
- **CCI_ER_BIND_ARRAY_SIZE**

cci_bind_param_array_size

Description

This function is used to determine the size of the array to be used in cci_bind_param_array(). **cci_bind_param_array_size**() must be called first before cci_bind_prarm_array() is used.

Syntax

```
int cci_bind_param_array_size(int req_handle, int array_size)
```

- *req_handle* : (IN) Request handle of a prepared statement
- *array_size* : (IN) Binding array size

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**

cci_blob_free

Description

This function frees memory of **BLOB** structure.

Syntax

```
int cci_blob_free (T_CCI_BLOB blob)
```

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_INVALID_LOB_HANDLE**

cci_blob_new

Description

This function creates an empty file where **LOB** data is stored and returns Locator referring to the data to *blob* structure.

Syntax

```
int cci_blob_new(int conn_handle, T_CCI_BLOB* blob, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *blob* : (OUT) **LOB** Locator
- *error_buf* : (OUT) Error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**
- **CCI_ER_INVALID_HANDLE**

cci_blob_write

Description

This function reads **LOB** data as long as the value of *length* from *start_pos* in **LOB** data file, stores the value in *buf*, and then returns it.

Syntax

```
int cci_blob_read(int conn_handle, T_CCI_BLOB blob, long start_pos, int length, const char *buf, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *blob* : (IN) **LOB** Locator
- *start_pos* : (IN) Index location of **LOB** data file
- *length* : (IN) **LOB** data length from buffer
- *error_buf* : (OUT) Error buffer

Return Value

- Size of read value (> =0 : success)
- Error code (< 0 : error)

Error Code

- **CCI_ER_INVALID_LOB_READ_POS**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**
- CCI_ER_INVALID_LOB_HANDLE

cci_blob_size

Description

This function returns data file size that is specified in *blob*.

Syntax

```
long long cci_blob_size (T_CCI_BLOB* blob)
```

- *blob* : (IN) **LOB** Locator

Return Value

- Size of **BLOB** data file (> =0 : success)
- Error code (<0 : error)

Error Code

- CCI_ER_INVALID_LOB_HANDLE

cci_blob_write

Description

This function reads data as long as the value of *length* from *buf* and then stores the value from *start_pos* in **LOB** data file.

Syntax

```
int cci_blob_write(int conn_handle, T_CCI_BLOB blob, long start_pos, int length, const char *buf, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *blob* : (IN) **LOB** Locator

- *start_pos* : (IN) Index location of **LOB** data file
- *length* : (IN) Data length from buffer
- *error_buf* : (OUT) Error buffer

Return Value

- Size of written value (> =0 : success)
- Error code (<0 : error)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**
- CCI_ER_INVALID_LOB_HANDLE

cci_clob_free

Description

This function frees memory of **CLOB** structure.

Syntax

```
int cci_clob_free (T_CCI_CLOB clob)
```

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_INVALID_LOB_HANDLE**

cci_clob_new

Description

This function creates an empty file where **LOB** data is stored and returns Locator referring to the data to *clob* structure.

Syntax

```
int cci_clob_new(int conn_handle, T_CCI_CLOB* clob, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *clob* : (OUT) **LOB** Locator
- *error_buf* : (OUT) Error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**

- **CCI_ER_INVALID_HANDLE**

cci_blob_write

Description

This function reads **LOB** data as long as the value of *length* from *start_pos* in **LOB** data file, stores the value in *buf*, and then returns it.

Syntax

```
int cci_clob_read(int conn_handle, T_CCI_CLOB clob, long start_pos, int length, const char *buf, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *clob* : (IN) **LOB** Locator
- *start_pos* : (IN) Index location of **LOB** data file
- *length* : (IN) **LOB** data length from buffer
- *error_buf* : (OUT) Error buffer

Return Value

- Size of read value (> =0 : success)
- Error code (< 0 : error)

Error Code

- **CCI_ER_INVALID_LOB_READ_POS**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**
- CCI_ER_INVALID_LOB_HANDLE

cci_clob_size

Description

This function returns data file size that is specified in *clob*.

Syntax

```
long long cci_clob_size (T_CCI_BLOB* clob)
```

- *clob* : (IN) **LOB** Locator

Return Value

- Size of **CLOB** data file (> =0 : success)
- Error code (<0 : error)

Error Code

- CCI_ER_INVALID_LOB_HANDLE

cci_clob_write

Description

This function reads data as long as the value of *length* from *buf* and then stores the value from *start_pos* in **LOB** data file.

Syntax

```
int cci_clob_write(int conn_handle, T_CCI_BLOB clob, long start_pos, int length, const
char *buf, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *clob* : (IN) **LOB** Locator
- *start_pos* : (IN) Index location of **LOB** data file
- *length* : (IN) Data length from buffer
- *error_buf* : (OUT) Error buffer

Return Value

- Size of written value (> =0 : success)
- Error code (<0 : error)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**
- CCI_ER_INVALID_LOB_HANDLE

cci_close_req_handle

Description

This function is used to close the request handle obtained by cci_prepare().

Syntax

```
int cci_close_req_handle(int req_handle)
```

- *req_handle* : (IN) Request handle

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_COMMUNICATION**

cci_col_get

Description

This function is used to get an attribute value of collection type. If the name of the class is C, and the domain of set_attr is set (multiset, sequence), the query looks like as follows:

```
SELECT a FROM C, TABLE(set_attr) AS t(a) WHERE C = oid;
```

That is, the number of members becomes the number of records.

Syntax

```
intcci_col_get (int conn_handle, char *oid_str, char *col_attr, int *col_size, int
*col_type, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *col_size* : (OUT) Collection size (-1 : null)
- *col_type* : (OUT) Collection type (set, multiset, sequence : u_type)
- *err_buf* : (OUT) Database error buffer

Return Value

- Request handle

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_seq_drop

Description

This function is used to drop the index-th (base:1) member of the sequence attribute values. The following is an example of dropping the first member of the sequence attribute values.

```
cci_col_seq_drop(con_id, oid_str, seq_attr, 1, err_buf);
```

Syntax

```
intcci_col_seq_drop (int conn_handle, char *oid_str, char *col_attr, int index,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *index* : (IN) Index
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_seq_insert

Description

This function is used to insert a member at the index-th (base:1) position of the sequence attribute values. The following is an example of inserting "a" at the first position of the sequence attribute values.

```
cci_col_seq_insert(con_id, oid_str, seq_attr, 1, "a", err_buf);
```

Syntax

```
intcci_col_seq_insert (int conn_handle, char *oid_str, char *col_attr, int index, char
*value, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *index* : (IN) Index
- *value* : (IN) Sequential element (string)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_seq_put

Description

This function is used to replace the index-th (base:1) member of the sequence attribute values with a new value. The following is an example of replacing the first member of the sequence attributes values with "a".

```
cci_col_seq_put(con_id, oid_str, seq_attr, 1, "a", err_buf);
```

Syntax

```
intcci_col_seq_put (int conn_handle, char *oid_str, char *col_attr, int index, char *value,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- col_attr : (IN) Collection attribute name
- index : (IN) Index
- value : (IN) Sequential value
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**

- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_set_add

Description

This function is used to add a member to the set attribute values. The following is an example of adding "a" to the set attribute values.

```
cci_col_set_add(con_id, oid_str, set_attr, "a", err_buf);
```

Syntax

```
intcci_col_set_add ( int conn_handle, char *oid_str, char *col_attr, char *value,
T_CCI_ERRROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *value* : (IN) Set element
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_set_drop

Description

This function is used to drop a member from the set attribute values. The following is an example of dropping "a" from the set attribute values.

```
cci_col_set_drop(con_id, oid_str, set_attr, "a", err_buf);
```

Syntax

```
intcci_col_set_drop (int conn_handle, char *oid_str, char *col_attr, char *value,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *value* : (IN) Set element (string)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

cci_col_size

Description

This function is used to get the size of the set (seq) attribute.

Syntax

```
intcci_col_size (int conn_handle, char *oid_str, char *col_attr, int *col_size,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *col_size* : (OUT) Collection size (-1 : NULL)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_connect

Description

A connection handle to the database server is assigned and it tries to connect to the server. If it has succeeded, the connection handle ID is returned; if fails, an error code is returned.

Syntax

```
int cci_connect(char *ip, int port, char *db_name, char *db_user, char *db_password)
```

- *ip* : (IN) A character string representing the IP address of the server (host name)
- *port* : (IN) Broker port (the port configured in the **$CUBRID/conf/cubrid_broker.conf** file)
- *db_name* : (IN) Database name
- *db_user* : (IN) Database user name
- *db_passwd* : (IN) Database user password

Return Value

- Success : Connection handle ID (int)
- Failure : Error code

Error Code

- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_HOSTNAME**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_connect_with_url

Description

cci_connect_with_url tries to connect a database by using connection information passed with an url string argument. If the HA feature is enabled in CCI, you must specify the connection information of the standby server, which is used for failover when failure occurs, in the url string argument of this function. If it has succeeded, the ID of connection handle is returned; if it fails, an error code is returned.

Syntax

```
int cci_connect_with_url (char *url [, char *db_user, char *db_password ])

<url> ::=
cci:CUBRID:<host>:<db_name>:<db_user>:<db_password>:[?<properties>]

<properties> ::= <property> [&<property>]
<property> ::= althosts=<alternative_hosts> [&rctime=<time>]
<alternative_hosts> ::= <standby_broker1_host>:<port> [,<standby_broker2_host>:<port>]

<host> := HOSTNAME | IP_ADDR
<time> := SECOND
```

- *url* : (IN) A character string that contains server connection information
- *host* : A host name or IP address of the master database
- *db_name* : A name of the database
- *db_user* : A name of the database user
- *db_password* : A database user password
- **althosts** =*standby_broker1_host, standby_broker2_host, . . .* : Specifies the broker information of the standby server, which is used for failover when it is impossible to connect to the active server. You can specify multiple brokers for failover, and the connection to the brokers is attempted in the order listed in **alhosts**.
- **rctime** : An interval between the attempts to connect to the active broker in which failure occurred. After a failure occurs, the system connects to the broker specified by althosts (failover), terminates the transaction, and then attempts to connect to the active broker of the master database at every **rctime**. The default value is 600 seconds.
- *db_user* : (IN) A name of the database user
- *db_passwd* : (IN) A database user password

Return Value

- Success : Connection handle ID (int)
- Failure : Error code

Error Code

- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_HOSTNAME**
- **CCI_ER_INVALID_URL**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

Example

```
--connection URL string when a property(althosts) specified for HA
URL=cci:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000

--connection URL string when properties(althosts,rctime) specified for HA
URL=cci:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600
```

cci_cursor

Description

This function is used to move the cursor specified in the request handle to access the specific record in the query result executed by cci_execute(). The position of cursor is moved by the values specified in the *origin* and *offset* values. If the position to be moved is not valid, **CCI_ER_NO_MORE_DATA is** returned.

Syntax

```
int cci_cursor(int req_handle, int offset, T_CCI_CURSOR_POS origin, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle
- *offset* : (IN) Offset to be moved
- *origin* : (IN) Variable to represent a position. The type is **T_CCI_CURSOR_POS**. **T_CCI_CURSOR_POS** enum consists of **CCI_CURSOR_FIRST**, **CCI_CURSOR_CURRENT**, and **CCI_CURSOR_LAST**.
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_NO_MORE_DATA**
- **CCI_ER_COMMUNICATION**

Example

```
//the cursor moves to the first record
cci_cursor(req, 1, CCI_CURSOR_FIRST, &err_buf);

//the cursor moves to the next record
cci_cursor(req, 1, CCI_CURSOR_CURRENT, &err_buf);

//the cursor moves to the last record
cci_cursor(req, 1, CCI_CURSOR_LAST, &err_buf);

//the cursor moves to the previous record
cci_cursor(req, -1, CCI_CURSOR_CURRENT, &err_buf);
```

cci_cursor_update

Description

This function is used to update *cursor_pos* from the value of the *index* th column to *value* . If the database is updated to **NULL**, *value* becomes **NULL**. For update conditions, see cci_prepare(). The data type of *value* for *a_type* is shown in the table below.

a_type	**value Type**
CCI_A_TYPE_STR	char*
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	**T_CCI_BIT***
CCI_A_TYPE_SET	**T_CCI_SET**

CCI_A_TYPE_DATE	**T_CCI_DATE***
CCI_A_TYPE_BIGINT	int64_t (For Windows : __int64)
CCI_A_TYPE_BLOB T_CCI_BLOB	
CCI_A_TYPE_CLOB T_CCI_CLOB	

Syntax

```
int cci_cursor_update(int req_handle, int cursor_pos, int index, T_CCI_A_TYPE a_type, void
*value, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle
- *cursor_pos* : (IN) Cursor position
- *index* : (IN) Column index
- *a_type* : (IN) *value* Type
- *value* : (IN) A new value
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : no error)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_ATYPE**

cci_disconnect

Description

This function is used to disconnect all request handles created for *conn_handle*. If a transaction is being performed, the handles are disconnected after cci_end_tran() is executed.

Syntax

```
int cci_disconnect(int conn_handle, T_CCI_ERROR * err_buf)
```

- *conn_handle* : (IN) Connection handle
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

cci_end_tran

Description

This function is used to perform a commit or rollback on the current transaction. At this point, all open request handles are terminated and the connection to the database server is disabled. However, even after the connection to the server is disabled, the connection handle remains valid. This is the same state as one in which one connection handle has been assigned by the cci_connect() function. The transaction is committed if the type is set to **CCI_TRAN_COMMIT**; and is rolled back if it is set to **CCI_TRAN_ROLLBACK**.

Syntax

```
int cci_end_tran(int conn_handle, char type, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *type* : (IN) **CCI_TRAN_COMMIT** or **CCI_TRAN_ROLLBACK**
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_TRAN_TYPE**

Remark

Auto-commit mode is supported for SELECT statements. To apply this mode, you must add **SELECT_AUTO_COMMIT=ON** to the cubrid_broker.conf file. However, auto-commit is performed only at the point at which the result set for all n query statements is fetched from the server when there are n prepared statements. An example is as follows:

Example 1

```
$sql1 = "select * from db_user";
$sql2 = "select * from db_class where owner_name = ?";

$result = cubrid_execute($con, $sql1);  // 1 select handle. fetch completed - autocommit
if ($result) {
  while ($row = cubrid_fetch ($result))
    {
      echo ($row[0]);

      $req = cubrid_prepare ($con, $sql2);
      cubrid_bind ($req, 1, $row[0]);
      $res = cubrid_execute ($req);   // 1 select handle. fetch completed - autocommit
    }
}
```

Example 2

```
$sql1 = "select * from db_user";
$sql2 = "select * from db_class where owner_name = ?";

$req = cubrid_prepare ($con, $sql2);
$result = cubrid_execute($con, $sql1);  // 2 handle. fetch completed for only 1 hanlde -
no autcommit
if ($result) {
  while ($row = cubrid_fetch ($result))
    {
      echo ($row[0]);

      cubrid_bind ($req, 1, $row[0]);
      $res = cubrid_execute ($req);   // fetch completed for all select handles -
autocommit
    }
}
```

Example 3

```
$sql1 = "select * from db_user";
$sql2 = "insert into a values (?)";

$result = cubrid_execute($con, $sql1);  // 1 select handle. fetch completed - autocommit
if ($result) {
  while ($row = cubrid_fetch ($result))
    {
      echo ($row[0]);

      $req = cubrid_prepare ($con, $sql2);
      cubrid_bind ($req, 1, $row[0]);
      $res = cubrid_execute ($req);    // no autocommit for insert
    }
}
```

Example 4

```
$sql1 = "select * from db_user";
$sql2 = "insert into a values (?)";

$req = cubrid_prepare ($con, $sql2);
$result = cubrid_execute($con, $sql1);  // no autocommit for insert because no fetch
if ($result) {
  while ($row = cubrid_fetch ($result))
    {
      echo ($row[0]);

      cubrid_bind ($req, 1, $row[0]);
      $res = cubrid_execute ($req);   // no autocommit for insert
    }
}
```

cci_execute

Description

This function is used to execute the prepared SQL statement, which is executing cci_prepare(). A request handle, a *flag*, the maximum length of the column to be fetched and the address of the **T_CCI_ERROR** construct to contain the error information are specified as parameters for this function.

The function of retrieving the query result from the server through a *flag* can be classified as synchronous or asynchronous. If the flag is set to **CCI_EXEC_QUERY_ALL**, a synchronous mode (sync_mode) is used to retrieve query results immediately after executing prepared queries if it is set to **CCI_EXEC_ASYNC**, an asynchronous mode (async_mode) is used to retrieve the result immediately each time a query result is created. The *flag* is set to **CCI_EXEC_QUERY_ALL** by default, and in such cases the following rules are applied.

- The return value is the result of the first query.
- If an error occurs in any query, the execution is processed as a failure.
- For a query composed of in a query composed of q1 q2 q3 if an error occurs in q2 after q1 succeeds the execution, the result of q1 remains valid. That is, the previous successful query executions are not rolled back when an error occurs.
- If a query is executed successfully, the result of the second query can be obtained using cci_next_result().

max_col_size is a value that is used to determine the size of the column to be transferred to the client when the type of the column of the prepared query is **CHAR**, **VARCHAR**, **NCHAR**, **VARNCHAR**, **BIT** or **VARBIT**. If it is set to 0, all data is transferred.

Syntax

```
int cci_execute(int req_handle, char flag, int max_col_size, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *flag* : (IN) Exec flag (**CCI_EXEC_ASYNC** or **CCI_EXEC_QUERY_ALL**)
- *max_col_size* : (IN) The size of the column to be fetched

- *err_buf* : (OUT) Database error buffer

Return Value

- Success
- **SELECT** : Returns the number of results in sync mode returns 0 in async mode.
- **INSERT**, **UPDATE** : Returns the number of tuples reflected.
- Others queries : 0
- Failure : Error code

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_BIND**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

cci_execute_array

Description

If more than one value are bound to the prepared statement, this gets the values of the variables to be bound and executes the query by binding each value to the variable.

To bind the data, call the cci_bind_param_array_size() function to specify the size of the array, bind each value to the variable by using the cci_bind_param_array() function, and execute the query by calling the **cci_execute_array**() function.

You can get three execution results by calling the cci_execute() function. However, the **cci_execute_array**() function returns the number of queries executed by the query_result variable. You can use the following macro to get the information about the execution result. However, note that the validity check is not performed for each parameter entered in the macro. After using the query_result variable, you must delete the query_result by using the cci_query_result_free() function.

Marco	Return Type	Meaning
CCI_QUERY_RESULT_RESULT	int	the number of results
CCI_QUERY_RESULT_ERR_MSG	char*	error message about query
CCI_QUERY_RESULT_STMT_TYPE	int(T_CCI_CUBRID_STMT enum)	type of query statement

Syntax

```
int cci_execute_array(int req_handle, T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR
*err_buf)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *query_result* : (OUT) Query results (the number of executed queries)
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : The number of executed queries
- Failure : Negative number

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_BIND**

- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

Example

```
char *query =
  "update participant set gold = ? where host_year = ? and nation_code = 'KOR'";
int gold[2];
char *host_year[2];
int null_ind[2];
T_CCI_QUERY_RESULT *result;
int n_executed;
...

req = cci_prepare (con, query, 0, &cci_error);
if (req < 0)
{
  printf ("prepare error: %d, %s\n", cci_error.err_code, cci_error.err_msg);
  goto handle_error;
}

gold[0] = 20;
host_year[0] = "2004";

gold[1] = 15;
host_year[1] = "2008";

null_ind[0] = null_ind[1] = 0;
error = cci_bind_param_array_size (req, 2);
if (error < 0)
{
  printf ("bind_param_array_size error: %d\n", error);
  goto handle_error;
}

error =
  cci_bind_param_array (req, 1, CCI_A_TYPE_INT, gold, null_ind, CCI_U_TYPE_INT);
if (error < 0)
{
  printf ("bind_param_array error: %d\n", error);
  goto handle_error;
}
error =
  cci_bind_param_array (req, 2, CCI_A_TYPE_STR, host_year, null_ind, CCI_U_TYPE_INT);
if (error < 0)
  {
  printf ("bind_param_array error: %d\n", error);
  goto handle_error;
}

n_executed = cci_execute_array (req, &result, &cci_error);
if (n_executed < 0)
{
  printf ("execute error: %d, %s\n", cci_error.err_code,
          cci_error.err_msg);
  goto handle_error;
}
for (i = 1; i <= n_executed; i++)
{
  printf ("query %d\n", i);
  printf ("result count = %d\n", CCI_QUERY_RESULT_RESULT (result, i));
  printf ("error message = %s\n", CCI_QUERY_RESULT_ERR_MSG (result, i));
  printf ("statement type = %d\n",
          CCI_QUERY_RESULT_STMT_TYPE (result, i));
}
error = cci_query_result_free (result, n_executed);
if (error < 0)
{
  printf ("query_result_free: %d\n", error);
  goto handle_error;
}
error = cci_end_tran(con, CCI_TRAN_COMMIT, &cci_error);
```

```
if (error < 0)
{
  printf ("end_tran: %d, %s\n", cci_error.err_code, cci_error.err_msg);
  goto handle_error;
}
```

cci_execute_batch

Description

In CCI, multiple jobs can be processed simultaneously when using DML queries such as **INSERT**/**UPDATE**/**DELETE**. cci_execute_array() and **cci_execute_batch**() functions can be used to execute such batch jobs. Note that prepared statements cannot be used in the **cci_execute_batch**() function.

Executes sql_stmt as many times as num_sql_stmt specified as a parameter and returns the number of queries executed with the query_result variable. You can use the macro (CCI_QUERY_RESULT_RESULT, CCI_QUERY_RESULT_ERR_MSG, CCI_QUERY_RESULT_STMT_TYPE) available in the cci_execute_array() function to get the information about the execution result.

However, note that the validity check is not performed for each parameter entered in the macro. After using the *query_result* variable, you must delete the query result by using the cci_query_result_free() function.

Syntax

```
int cci_execute_batch(int conn_handle, int num_sql_stmt, char **sql_stmt,
T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *num_sql_stmt* : (IN) The number of *sql_stmt*s
- *sql_stmt* : (IN) SQL statement array
- *query_result* : (OUT) The results of *sql_stmt*
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : The number of executed queries
- Failure : Negative number

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_CONNECT**

Example

```
char **queries;
  T_CCI_QUERY_RESULT *result;
  int n_queries, n_executed;
...

  count = 3;
  queries = (char **) malloc (count * sizeof (char *));
  queries[0] =
    "insert into athlete(name, gender, nation_code, event) values('Ji-sung Park', 'M',
'KOR', 'Soccer')";
  queries[1] =
    "insert into athlete(name, gender, nation_code, event) values('Joo-young Park', 'M',
'KOR', 'Soccer')";
  queries[2] =
    "select * from athlete order by code desc for orderby_num() < 3";
//calling cci_execute_batch()
```

```
  n_executed = cci_execute_batch (con, count, queries, &result, &cci_error);
  if (n_executed < 0)
    {
      printf ("execute_batch: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }
  printf ("%d statements were executed.\n", n_executed);

  for (i = 1; i <= n_executed; i++)
    {
      printf ("query %d\n", i);
      printf ("result count = %d\n", CCI_QUERY_RESULT_RESULT (result, i));
      printf ("error message = %s\n", CCI_QUERY_RESULT_ERR_MSG (result, i));
      printf ("statement type = %d\n",
              CCI_QUERY_RESULT_STMT_TYPE (result, i));
    }

  error = cci_query_result_free (result, n_executed);
  if (error <
0)

    {

      printf ("query_result_free: %d\n", error);
      goto handle_error;
    }
```

cci_execute_result

Description

This function is used to get the execution results (e.g. statement type, result count) performed by cci_execute(). The results of each query are retrieved by CCI_QUERY_RESULT_STMT_TYPE and CCI_QUERY_RESULT_RESULT. The query results used must be deleted by cci_query_result_free.

Syntax

```
int cci_execute_result(int req_handle, T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR
*err_buf)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *query_result* : (OUT) Query results
- *err_buf* : (OUT) Database error buffer

Return Value

- Suceess : The number of queries
- Failure : Negative number

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_COMMUNICATION**

Example

```
T_CCI_QUERY_RESULT *qr;
…

cci_execute( … );
res = cci_execute_result(req_h, &qr, &err_buf);
if (res < 0) {
  /* error */
}
else {
  for (i=1 ; i <= res ; i++) {
```

```
    result_count = CCI_QUERY_RESULT_RESULT(qr, i);
    stmt_type = CCI_QUERY_RESULT_STMT_TYPE(qr, i);
  }
  cci_query_result_free(qr, res);
}
```

cci_fetch

Description

Fetches the query result executed by cci_execute() from the server-side CAS and saves it to the client buffer. The cci_get_data() function can be used to identify the data of a specific column from the fetched query result.

Syntax

```
int cci_fetch(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

cci_fetch_buffer_clear

Description

This function is used to clear the records temporarily saved in the client buffer.

Syntax

```
int cci_fetch_buffer_clear(int req_handle)
```

- *req_handle* : (IN) Request handle

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**

cci_fetch_sensitive

Description

This function is used to send changed values for sensitive columns when the results are sent to the client from the server. If the results by *req_handle* are not sensitive, they are same as the ones by cci_fetch(). The return value of **CCI_ER_DELETED_TUPLE** means that the given tuple has been deleted.

Syntax

```
int cci_fetch_sensitive(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_NO_MORE_DATA**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_DBMS**
- **CCI_ER_DELETED_TUPLE**

cci_fetch_size

Description

This function is used to determine the number of records sent by cci_fetch() from the server to the client.

Syntax

```
int cci_fetch_size(int req_handle, int fetch_size)
```

- *req_handle :* (IN) Request handle
- *fetch_size* : (IN) Fetch size

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**

cci_get_autocommit

Description

The **cci_get_autocommit**() function returns the auto-commit mode currently configured.

Syntax

```
int cci_get_autocommit (int conn_handle)
```

- *conn_handle* : Connection handle

Return Value

- 1 : Auto-commit ON
- 0 : Auto-commit OFF

Error Code

- None

cci_get_bind_num

Description

This function is used to get the number of input bindings. If the SQL statement used during preparation is composed of multiple queries, it represents the number of input bindings used in all queries.

Syntax

```
int cci_get_bind_num(int req_handle)
```

- *req_handle* : (IN) Request handle for a prepared SQL statement

Return Value

- The number of input bindings

Error Code

- **CCI_ER_REQ_HANDLE**

cci_get_class_num_objs

Description

This function is used to get the number of objects of the *class_name* class and the number of pages being used. If the flag is configured to 1, an approximate value is fetched; if it is configured to 0, an exact value is fetched.

Syntax

```
int cci_get_class_num_objs(int conn_handle, char *class_name, int flag, int *num_objs, int *num_pages, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *class_name* : (IN) Class name
- *flag* : (IN) 0 or 1
- *num_objs* : (OUT) The number of objects
- *num_pages* : (OUT) The number of pages
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

CCI_GET_COLLECTION_DOMAIN

Description

If *u_type* is set, multiset or sequence type, this macro gets the domain of the set, multiset or sequence. If *u_type* is not a set type, the return value is the same as *u_type*.

Syntax

```
#define CCI_GET_COLLECTION_DOMAIN(u_type)
```

Return Value

- Type (CCI_U_TYPE)

cci_get_cur_oid

Description

This function is used to get the OID of the currently fetched records if **CCI_INCLUDE_OID** is configured in execution. The OID is represented as a string for a page, slot or volume.

Syntax

```
intcci_get_cur_oid(int req_handle, char *oid_str_buf)
```

- *conn_handle :* (IN) Request handle
- *oid_str_buf* : (OUT) OID string

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**

cci_get_data

Description

Gets the *col_noth* value from the currently fetched result. The *type* of the *value* variable is determined according to the given *type* parameter, and the value or the pointer is copied to the value variable accordingly.

For a value to be copied, the memory for the address to be transferred to the *value* variable must have been previously assigned. Note that if a pointer is copied, a pointer in the application client library is returned, so the value becomes invalid next time the **cci_get_data**() function is called.

In addition, the pointer returned by the pointer copy must not be freed. However, if the type is **CCI_A_TYPE_SET**, the memory must be freed by using the cci_set_free() function after using the set because the set is returned after the **T_CCI_SET** type memory is allocated. The following table shows the summary of *type* parameters and data types of their corresponding *values*.

type	value Type	Meaning
CCI_A_TYPE_STR	char**	pointer copy
CCI_A_TYPE_INT	int*	value copy
CCI_A_TYPE_FLOAT	float*	value copy
CCI_A_TYPE_DOUBLE	double*	value copy
CCI_A_TYPE_BIT	**T_CCI_BIT***	value copy (pointer copy for each member)
CCI_A_TYPE_SET	**T_CCI_SET***	memory alloc and value copy
CCI_A_TYPE_DATE	**T_CCI_DATE***	value copy
CCI_A_TYPE_BIGINT	int64_t* (For Windows : __int64*)	value copy
CCI_A_TYPE_BLOB	**T_CCI_BLOB**	memory alloc and value copy
CCI_A_TYPE_CLOB	**T_CCI_CLOB**	memory alloc and value copy

Syntax

```
int cci_get_data(int req_handle, int col_no, int type, void *value, int *indicator)
```

- *req_handle* : (IN) Request handle
- *col_no* : (IN) One-based column index. It starts with 1.
- *type* : (IN) Data type (defined in the **T_CCI_A_TYPE**) of *value* variable
- *value* : (OUT) Variable address for data to be stored
- *indicator* : (OUT) **NULL** indicator (-1 : **NULL**)
- if *type* is **CCI_A_TYPE_STR** : -1 is returned in case of **NULL**; the length of character string stored in *value* is returned, otherwise.
- if *type* is **CCI_A_TYPE_STR** : -1 is returned in case of **NULL**, 0 is returned, otherwise.

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_COLUMN_INDEX**
- **CCI_ER_ATYPE**

cci_get_db_parameter

Description

This function is used to get a parameter specified in the database. The data type of *value* for *param_name* is shown in the table below.

param_name	value Type	note
CCI_PARAM_ISOLATION_LEVEL	int*	get/set
CCI_PARAM_LOCK_TIMEOUT	int*	get/set
CCI_PARAM_MAX_STRING_LENGTH	int*	get only

Syntax

```
int cci_get_db_parameter(int conn_handle, T_CCI_DB_PARAM param_name, void *value,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *param_name* : (IN) System parameter name
- *value* : (OUT) Parameter value
- *err_buf* : (OUT) Database error buffer

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_PARAM_NAME**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_get_db_version

Description

This function is used to get the Database Management System (DBMS) version.

Syntax

```
int cci_get_db_version(int conn_handle, char *out_buf, int out_buf_size)
```

- *conn_handle* : (IN) Connection handle
- *out_buf* : (OUT) Result buffer
- *out_buf_size* : (IN) *oub_buf* size

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_get_result_info

Description

If the prepared statement is **SELECT**, the **T_CCI_COL_INFO** struct that stores the column information about the execution result can be obtained by using this function. If it is not **SELECT**, **NULL** is returned and the *num* value becomes 0.

You can access the **T_CCI_COL_INFO** struct directly to get the column information from the struct, but you can also use a macro to get the information, which is defined as follows. The address of the **T_CCI_COL_INFO** struct and the column index are specified as parameters for each macro. The macro can be called only for the **SELECT** query. Note that the validity check is not performed for each parameter entered in each macro. If the return type of the macro is char*, do not free the memory pointer.

Macro	Return Type	Meaning
CCI_GET_RESULT_INFO_TYPE	T_CCI_U_TYPE	column type
CCI_GET_RESULT_INFO_SCALE	short	column scale
CCI_GET_RESULT_INFO_PRECISION	int	column precision
CCI_GET_RESULT_INFO_NAME	char*	column name
CCI_GET_RESULT_INFO_ATTR_NAME	char*	column attribute name
CCI_GET_RESULT_INFO_CLASS_NAME	char*	column class name
CCI_GET_RESULT_INFO_IN_NON_NULL	char(0 or 1)	whether a column is NULL

Syntax

```
T_CCI_COL_INFO* cci_get_result_info(int req_handle, T_CCI_SQLX_CMD *cmd_type, int *num)
```

- *req_handle* : (IN) Request handle for a prepared SQL statement
- *cmd_type* : (OUT) Command type
- *num* : (OUT) The number of columns in the **SELECT** statement (if *cmd_type* is **SQLX_CMD_SELECT**)

Return Value

- Success : Result info pointer
- Failure : **NULL**

Example

```
col_info = cci_get_result_info (req, &cmd_type, &col_count);
  if (col_info == NULL)
    {
      printf ("get_result_info error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }
  for (i = 1; i <= col_count; i++)
    {
      printf ("%-12s = %d\n", "type", CCI_GET_RESULT_INFO_TYPE (col_info, i));
      printf ("%-12s = %d\n", "scale",
              CCI_GET_RESULT_INFO_SCALE (col_info, i));
      printf ("%-12s = %d\n", "precision",
              CCI_GET_RESULT_INFO_PRECISION (col_info, i));
```

```
    printf ("%-12s = %s\n", "name", CCI_GET_RESULT_INFO_NAME (col_info, i));
    printf ("%-12s = %s\n", "attr_name",
            CCI_GET_RESULT_INFO_ATTR_NAME (col_info, i));
    printf ("%-12s = %s\n", "class_name",
            CCI_GET_RESULT_INFO_CLASS_NAME (col_info, i));
    printf ("%-12s = %s\n", "is_non_null",
            CCI_GET_RESULT_INFO_IS_NON_NULL (col_info,i) ? "true" : "false");
```

CCI_GET_RESULT_INFO_ATTR_NAME

Description

This macro is used to get the actual attribute name of the *index* -th column of a prepared **SELECT** statement. If there is no name for the attribute (constant, function, etc), " " (empty string) is returned. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid. You cannot delete the returned memory pointer with **free**().

Syntax

```
#define CCI_GET_RESULT_INFO_ATTR_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) pointer to the column information fetched by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Attribute name (char*)

CCI_GET_RESULT_INFO_CLASS_NAME

Description

This macro is used to get the *index* -th class name of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid. You cannot delete the returned memory pointer with **free**(). The returned value can be **NULL**.

Syntax

```
#define CCI_GET_RESULT_INFO_CLASS_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Class name (char*)

CCI_GET_RESULT_INFO_IS_NON_NULL

Description

This macro is used to get a value indicating whether the *index* -th column of a prepared **SELECT** statement is nullable. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_GET_RESULT_INFO_IS_NON_NULL(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer by cci_get_result_info
- *index* : (IN) Column index

Return Value

- 0 : nullable

- 1 : non **NULL**

CCI_GET_RESULT_INFO_NAME

Description

This macro is used to get the *index* -th column name of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid. You cannot delete the returned memory pointer with **free**().

Syntax

```
#define CCI_GET_RESULT_INFO_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer to cci_get_result_info
- *index* : (IN) Column index

Return Value

- Column name (char*)

CCI_GET_RESULT_INFO_PRECISION

Description

This macro is used to get the *index* -th precision of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_GET_RESULT_INFO_PRECISION(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Precision (int)

CCI_GET_RESULT_INFO_SCALE

Description

This macro is used to get the *index* -th column's scale of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_GET_RESULT_INFO_SCALE(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Scale (int)

CCI_GET_RESULT_INFO_TYPE

Description

This macro is used to get the *index* -th column type of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_GET_RESULT_INFO_TYPE(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) pointer to the column information fetched by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Column type (**T_CCI_U_TYPE**)

CCI_IS_SET_TYPE, CCI_IS_MULTISET_TYPE, CCI_IS_SEQUENCE_TYPE, CCI_IS_COLLECTION_TYPE

Description

This macro is used to check whether *u_type* is set, multiset or sequence type.

Syntax

```
#define CCI_IS_SET_TYPE(u_type)
#define CCI_IS_MULTISET_TYPE(u_type)
#define CCI_IS_SEQUENCE_TYPE(u_type)
#define CCI_IS_COLLECTION_TYPE(u_type)
```

Return Value

- **CCI_IS_SET_TYPE**
- 1 : set
- 0 : not set
- **CCI_IS_MULTISET_TYPE**
- 1 : multiset
- 0 : not multiset
- **CCI_IS_SEQUENCE_TYPE**
- 1 : sequence
- 0 : not sequence
- **CCI_IS_SET_TYPE**
- 1 : collection (set, multiset, sequence)
- 0 : not collection

cci_is_updatable

Description

It is used to check whether the SQL statement, which executed cci_prepare(), is updatable. If it is updatable, 1 is returned.

Syntax

```
int cci_is_updatable(int req_handle)
```

- *req_handle* : (IN) Request handle for a prepared SQL statement

Return Value

- 1 : updatable
- 0 : not updatable

Error Code

- **CCI_ER_REQ_HANDLE**

cci_next_result

Description

The function is used to get results of next query if **CCI_EXEC_QUERY_ALL** *flag* is set upon cci_execute(). The information about the query fetched by next_result can be obtained with cci_get_result_info. If next_result is executed successfully, the database is updated with the information of the current query.

The error code **CAS_ER_NO_MORE_RESULT_SET** means that no more result set exists.

Syntax

```
int cci_next_result(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle of a prepared statement
- *err_buf* : (OUT) Database error buffer

Return Value

- Success
- **SELECT** (sync mode) : the number of results, (async mode) : 0
- **INSERT**, **UPDATE** : the number of records reflected
- Others : 0
- Failure : Error code

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

cci_oid

Description

CCI_OID_DROP : Deletes the given oid.

CCI_OID_IS_INSTANCE : Checks whether the given oid is an instance oid.

CCI_OID_LOCK_READ : Sets a read lock on the given oid.

CCI_OID_LOCK_WRITE : Sets a write lock on the given oid.

Syntax

```
intcci_oid(int conn_handle, T_CCI_OID_CMD cmd, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *cmd* : (IN) CCI_OID_DROP, CCI_OID_IS_INSTANCE, CCI_OID_LOCK_READ, CCI_OID_LOCK_WRITE
- *oid_str* : (IN) oid
- *err_buf* : (OUT) Database error buffer

Return Value

- CCI_OID_IS_INSTANCE
- 0 : non-instance
- 1 : instance
- < 0 : error
- CCI_OID_DROP, CCI_OID_LOCK_READ, CCI_OID_LOCK_WRITE
- Error code (0 : success, negative : failure)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OID_CMD**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_oid_get

Description

This function is used to get the attribute values of the given oid. *attr_name* is an array of the attributes, and it must end with **NULL**. If *attr_name* is NULL, the information of all attributes is fetched. The request handle has the same form as when the SQL statement "SELECT attr_name FROM oid_class WHERE oid_class = oid" is executed.

Syntax

```
int cci_oid_get(int conn_handle, char *oid_str, char **attr_name, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *attr_name* : (IN) A list of attributes
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : Request handle
- Failure : Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_CONNECT**

cci_oid_get_class_name

Description

This function is used to get the class name of the given oid.

Syntax

```
intcci_oid_get_class_name(int conn_handle, char *oid_str, char *out_buf, int out_buf_len,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *out_buf* : (OUT) Out buffer

- *out_buf_len* : (IN) *out_buf* length
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_oid_put

Description

This function is used to configure the *attr_name* attribute values of the given oid to *new_val_str*. The last value of *attr_name* must be **NULL**. Any value of any type must be represented as a string. The value represented as a string is applied to the database after being converted depending on the attribute type on the server. To insert a **NULL** value, configure the value of *new_val_str*[i] to **NULL**.

Syntax

```
intcci_oid_put(int conn_handle, char *oid_str, char **attr_name, char **new_val_str,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *attr_name* : (IN) A list of attribute names
- *new_val_str* : (IN) A list of new values
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**

cci_oid_put2

Description

This function is used to set the *attr_name* attribute values of the given oid to *new_val*. The last value of *attr_name* must be **NULL**. To insert a **NULL** value, set the value of *new_val*[i] to **NULL**.

The type of *new_val*[i] for *a_type* is shown in the table below.

Type of new_val[i] for a_type

Type	value type
CCI_A_TYPE_STR	char*
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*

CCI_A_TYPE_BIT	**T_CCI_BIT***
CCI_A_TYPE_SET	**T_CCI_SET**
CCI_A_TYPE_DATE	**T_CCI_DATE***
CCI_A_TYPE_BIGINT	int64_t (For Windows : __int64)

Syntax

```
intcci_oid_put2(int conn_handle, char *oidstr, char **attr_name, void **new_val, int
*a_type, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *attr_name* : (IN) A list of attribute names
- *new_val* : (IN) A new value array
- *a_type* : (IN) *new_val* type array
- *err_buf :* (OUT) Database error buffer

Return Value

- Error code (0 : success, negative number : failure)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**

Example

```
char *attr_name[array_size]
void *attr_val[array_size]
int a_type[array_size]
int int_val

…

attr_name[0] = "attr_name0"
attr_val[0] = &int_val
a_type[0] = CCI_A_TYPE_INT
attr_name[1] = "attr_name1"
attr_val[1] = "attr_val1"
a_type[1] = CCI_A_TYPE_STR

…
attr_name[num_attr] = NULL

res = cci_put2(con_h, oid_str, attr_name, attr_val, a_type, &error)
```

cci_prepare

Description

This function is used to prepare SQL execution by acquiring request handle for SQL statements. If a SQL statement consists of multiple queries, the preparation is performed only for the first query. With the parameter of this function, an address to **T_CCI_ERROR** where connection handle, SQL statement, *flag*, and error information are saved.

CCI_PREPARE_UPDATABLE or **CCI_PREPARE_INCLUDE_OID** can be configured in *flag*. If **CCI_PREPARE_UPDATABLE** is configured, updatable result set is created and **CCI_PREPARE_INCLUDE_OID** is automatically configured. However, not all updatable result sets are created even though **CCI_PREPARE_UPDATABLE** is configured. So you need to check if the results are updatable by using cci_is_updatable after preparation.

The conditions of updatable queries are as follows:

- A query must be **SELECT**.
- OID must be contained in the query result.
- The column to be updated must be the one that belongs to the table specified in the **FROM** clause.

Syntax

```
int cci_prepare(int conn_handle, char *sql_stmt, char flag,T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *sql_stmt* : (IN) SQL statement
- *flag* : (IN) prepare flag (**CCI_PREPARE_INCLUDE_OID** or **CCI_PREPARE_UPDATABLE**)
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : Request handle ID (int)
- Failure : Error code (negative)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_STR_PARAM**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_CONNECT**

CCI_QUERY_RESULT_ERR_MSG

Description

This macro is used to get error messages for the cci_execute_batch query. If there is no error message, " " (empty string) is returned. It does not check whether the specified argument, *query_result*, is **NULL**, and whether *index* is valid.

Syntax

```
#define CCI_QUERY_RESULT_ERR_MSG(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result* : (IN) Query results of cci_execute_batch
- *index* : (IN) Column index (base : 1)

Return Value

- Error message

cci_query_result_free

Description

This function is used to delete query result.

Syntax

```
int cci_query_result_free(T_CCI_QUERY_RESULT* query_result, int num_query)
```

- *query_result* : (IN) Query results of cci_execute_batch
- *num_query* : (IN) The number of arrays in *query_result*

Return Value

- Error code (0 : success, negative number : failure)

Example

```
T_CCI_QUERY_RESULT *qr;
char **sql_stmt;


res = cci_execute_array(conn, &qr, &err_buf);


cci_query_result_free(qr, res);
```

CCI_QUERY_RESULT_RESULT

Description

This macro is used to get the result count of the cci_execute_batch query. It does not check whether the specified argument, *query_result*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_QUERY_RESULT_RESULT(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result* : (IN) Query results of cci_execute_batch
- *index* : (IN) Column index (base : 1)

Return Value

- Result count

CCI_QUERY_RESULT_STMT_TYPE

Description

This macro is used to get the statement type of the cci_execute_batch query. It does not check whether the specified argument, *query_result*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_QUERY_RESULT_STMT_TYPE(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result* : (IN) Query results of cci_execute_batch
- *index* : (IN) Column index (base : 1)

Return Value

- Statement type (**T_CCI_SQLX_CMD**)

cci_savepoint

Description

This function is used to configure a savepoint or performs transaction rollback to a specified savepoint. Sets a savepoint if cmd is set to **CCI_SP_SET**. If it is set to **CCI_SP_ROLLBACK**, the transaction is rolled back to the specified savepoint.

Syntax

```
intcci_savepoint(int conn_handle, T_CCI_SAVEPOINT_CMD cmd, char* savepoint_name,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *cmd* : (IN) CCI_SP_SET or CCI_SP_ROLLBACK
- savepoint_name : (IN) Savepoint name

- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Example

```
con = cci_connect( ...);
.../* query execute */

/* sets a savepoint named "savepoint1"
cci_savepoint(con, CCI_SP_SET, "savepoint1", err_buf);

... /* query execute */

/* rolls back the set savepoint to "savepoint1" */
cci_savepoint(con, CCI_SP_ROLLBACK, "savepoint1", err_buf);
```

cci_schema_info

Description

This function is used to get schema information. If it is performed successfully, the results are managed by the request handle and can be fetched by fetch and getdata. If you want to retrieve a *class_name* of attr_name by pattern matching, configure the *flag*.

Two flags, **CCI_CLASS_NAME_PATTERN_MATCH** and **CCI_ATTR_NAME_PATTERN_MATCH**, are used for pattern matching. You can configure these two *flag*s by using the OR operator (|). Performance may significantly decrease if pattern matching is used.

The following table shows records composition of each *type*.

Record Composition of Each Type

Type	Column Order	Column Name	Column Type
CCI_SCH_CLASS	1	NAME	char*
	2	TYPE	short 0 : system class 1 : vclass 2 : class 3 : proxy
CCI_SCH_VCLASS	1	NAME	char*
	2	TYPE	short 1 : vclass 3 : proxy
CCI_SCH_ATTRIBUTE	1	NAME	char*
	2	DOMAIN	int
	3	SCALE	int
	4	PRECISION	int
	5	INDEXED	int 1 : indexed
	6	NON_NULL	int 1 : non null
	7	SHARED	int 1 : shared
	8	UNIQUE	int

			1 : unique
	9	DEFAULT	void*
	10	ATTR_ORDER	int base : 1
	11	CLASS_NAME	char*
	12	SOURCE_CLASS	char*
	13	IS_KEY	short 1 : key
CCI_SCH_CLASS_METHOD	1	NAME	char*
	2	RET_DOMAIN	int
	3	ARG_DOMAIN	char*
CCI_SCH_METHOD_FILE	1	METHOD_FILE	char*
CCI_SCH_super class	1	CLASS_NAME	char*
	2	TYPE	short
CCI_SCH_SUBCLASS	1	CLASS_NAME	char*
	2	TYPE	short
CCI_SCH_CONSTRAINT	1	TYPE 0 : unique 1 : index 2 : reverse unique 3 : reverse index	int
	2	NAME	char*
	3	ATTR_NAME	char*
	4	NUM_PAGES	int
	5	NUM_KEYS	int
	6	PRIMARY_KEY 1 : primary key	short
	7	KEY_ORDER	short base : 1
CCI_SCH_TRIGGER	1	NAME	char*
	2	STATUS	char*
	3	EVENT	char*
	4	TARGET_CLASS	char*
	5	TARGET_ATTR	char*
	6	ACTION_TIME	char*
	7	ACTION	char*
	8	PRIORITY	float
	9	CONDITION_TIME	char*
	10	CONDITION	char*
CCI_SCH_CLASS_PRIVILEGE	1	CLASS_NAME	char*
	2	PRIVELEGE	char*
	3	GRANTABLE	char*
CCI_SCH_ATTR_PRIVILEGE	1	ATTR_NAME	char*

	2	PRIVILEGE	char*
	3	GRANTABLE	char*
CCI_SCH_PRIMARY_KEY	1	CLASS_NAME	char*
	2	ATTR_NAME	char*
	3	KEY_SEQ	short base : 1
	4	KEY_NAME	char*
CCI_SCH_IMPORTED_KEY Used to retrieve primary key columns that are referred by a foreign key column in a given table. The results are sorted by PKTABLE_NAME and KEY_SEQ. If this type is specified as a parameter, a foreign key table is specified for *class_name*, and NULL is specified for *attr_name*.	1	PKTABLE_NAME	char**
	2	PKCOLUMN_NAME	char**
	3	FKTABLE_NAME	char**
	4	FKCOLUMN_NAME	char**
	5	KEY_SEQ	char**
	6	UPDATE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	7	DELETE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	8	FK_NAME	char**
	9	PK_NAME	char**
CCI_SCH_EXPORTED_KEYS Used to retrieve primary key columns that are referred by all foreign key columns. The results are sorted by FKTABLE_NAME and KEY_SEQ. If this type is specified as a parameter, a primary key table is specified for *class_name*, and NULL is specified for *attr_name*.	1	PKTABLE_NAME	char**
	2	PKCOLUMN_NAME	char**
	3	FKTABLE_NAME	char**
	4	FKCOLUMN_NAME	char**
	5	KEY_SEQ	char**
	6	UPDATE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	7	DELETE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	8	FK_NAME	char**
	9	PK_NAME	char**
CCI_SCH_CROSS_REFERENCE Used to retrieve foreign key information when primary keys and foreign keys in a given table are cross referenced. The results are sorted by FKTABLE_NAME and KEY_SEQ. If this type is specified as a parameter, a primary key is specified for *class_name*, and a foreign key	1	PKTABLE_NAME	char**
	2	PKCOLUMN_NAME	char**
	3	FKTABLE_NAME	char**
	4	FKCOLUMN_NAME	char**
	5	KEY_SEQ	char**

table is specified for *attr_name*.	6	UPDATE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	7	DELETE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	8	FK_NAME	char**
	9	PK_NAME	char**

Pattern match

CCI_SCH_TYPE	**Class name**	**ATTR_name**
CCI_SCH_CLASS (VCLASS)	O	none
CCI_SCH_ATTRIBUTE (CLASS ATTRIBUTE)	O	O
CCI_SCH_CLASS_PRIVILEGE	O	none
CCI_SCH_ATTR_PRIVILEGE	X	O
CCI_SCH_PRIMARY_KEY	O	none

If the pattern flag is not configured, exact string matching is used for the given class or attribute name. Therefore, there is no result if NULL is given. If the name of the class or attribute is NULL when the pattern flag is configured, the result is the same as when "%" is used.

Note TYPE column of CCI_SCH_CLASS and CCI_SCH_VCLASS : The proxy type is added. When used in OLEDB, ODBC or PHP, vclass is represented without distinguishing between proxy and vclass.

Syntax

```
int cci_schema_info(int conn_handle, T_CCI_SCHEMA_TYPE type, char *class_name, char
*attr_name, char flag, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *type* : (IN) Schema type
- *class_name* : (IN) Class name or NULL
- *attr_name* : (IN) Attribute name of NULL
- *flag* : (IN) Pattern matching flag (**CCI_CLASS_NAME_PATTERN_MACTH** or **CCI_CLASS_NAME_PATTERN_MATCH**)
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : Request handle
- Failure : Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_SCHEMA_TYPE**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_CONNECT**

cci_set_autocommit

Description

The **cci_set_autocommit**() function configures the auto-commit mode. When this function is called, concurrent transactions are committed.

Syntax

```
int cci_set_autocommit (int conn_handle, int autocommit_mode)
```

- *conn_handle* : (IN) Connection handle which cci_connect() or cci_connect_with_url() returns
- autocommit_mode : (IN) 1 if autocommit mode is on, 0 if autocommit mode is off.

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**

cci_set_db_parameter

Description

This function is used to configure a system parameter. For the type of *value* for *param_name*, see cci_get_db_parameter().

Syntax

```
int cci_set_db_parameter(int conn_handle, T_CCI_DB_PARAM param_name, void* value,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *param_name* : (IN) System parameter name
- *value* : (IN) Parameter value
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_PARAM_NAME**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_set_element_type

Description

This function is used to get the element type for the set fetched by CCI_A_TYPE_SET with cci_get_data().

Syntax

```
intcci_set_element_type(T_CCI_SET set)
```

- *set* : (IN) cci set pointer

Return Value

- Type

cci_set_free

Description

This function is used to release the memory assigned to **T_CCI_SET gotten** by **CCI_A_TYPE_SET** with cci_get_data().

Syntax

```
voidcci_set_free(T_CCI_SET set)
```

- *set* : (IN) cci set pointer

Return Value

- None

cci_set_get

Description

This function is used to get the index-th data for the set fetched by **CCI_A_TYPE_SET** with cci_get_data(). The data type of *value* for *a_type* is shown in the table below.

a_type	***value*** **Type**
CCI_A_TYPE_STR	char**
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	**T_CCI_BIT***
CCI_A_TYPE_DATE	**T_CCI_DATE***
CCI_A_TYPE_BIGINT	int64_t* (For Windows : __int64*)

Syntax

```
intcci_set_get(T_CCI_SET set, int index, T_CCI_A_TYPE a_type, void *value, int *indicator)
```

- *set* : (IN) cci set pointer
- *index* : (IN) Set index (base : 1)
- *a_type* : (IN) Type
- *value* : (OUT) Result buffer
- *indicator* : (OUT) Null indicator

Return Value

- Error code

cci_set_isolation_level

Description

This function is used to set the transaction isolation level of connections. All further transactions for the given connections work as *new_isolation_level*.

Note If the transaction isolation level is set by cci_set_db_parameter(), only the current transaction is affected. When the transaction is complete, the transaction isolation level returns to the one set by CAS. You must use **cci_set_isolation_level**() to set the isolation level for the entire connection.

Syntax

```
intcci_set_isolation_level(int conn_handle, T_CCI_TRAN_ISOLATION  new_isolation_level,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *new_isolation_level* : (IN) Transaction isolation level
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_ISOLATION_LEVEL**
- **CCI_ER_DBMS**

cci_set_make

Description

This function is used to make a set of a new **CCI_A_TYPE_SET** type. The created set is sent to the server as **CCI_A_TYPE_SET** by cci_bind_param(). The memory for the set created by **cci_set_make()** must be freed by **cci_set_free()**. The type of *value* for *u_type* is shown in the table below.

Syntax

```
intcci_set_make(T_CCI_SET *set, T_CCI_U_TYPE u_type, int size, void *value, int *indicator)
```

- set : (IN) cci set pointer
- *u_type* : (IN) Element type
- *size* : (IN) Set size
- *value* : (IN) Set element
- *indicator* : (IN) Null indicator array

Return Value

- Error code

cci_set_max_row

Description

This function is used to configure the maximum number of records for the results of the **SELECT** statement executed by cci_execute. If the *max* value is 0, it is the same as not setting the value.

Syntax

```
intcci_set_max_row(int req_handle, int max)
```

- *req_handle* : (IN) Connection handle
- *max* : (IN) The maximum number of rows

Return Value

- Error code

Example

```
req = cci_prepare( … );
cci_set_max_row(req, 1);
cci_execute( … );
```

cci_set_size

Description

This function is used to get the number of elements for the set fetched by **CCI_A_TYPE_SET** with cci_get_data().

Syntax

```
intcci_set_size(T_CCI_SET set)
```

- *set* : (IN) cci set pointer

Return Value

- Size