ALTIBASE® HDB™ Administration

Getting Started Guide

Release 6.5.1

May 28, 2015

ALTIBASE

ALTIBASE HDB Administration Getting Starting Guidel
Release 6.5.1

Altibase Corporation
10F, Daerung PostTower II, 182-13,
Guro-dong Guro-gu Seoul, 152-847, South Korea
Telephone: +82-2-2082-1000 Fax: 82-2-2082-1099
Homepage: http://www.altibase.com

Contents

Preface

About This Manual

This manual explains the concepts, components, and basic use of ALTIBASE HDB.

Audience

This manual has been prepared for the following ALTIBASE HDB users:

- database managers
- performance managers
- database users
- application developers
- technical support workers

It is recommended that those reading this manual possess the following background knowledge:

- basic knowledge in the use of computers, operating systems, and operating system utilities
- experience in using relational databases and an understanding of database concepts
- computer programming experience
- experience in database server, operating system or network administration

Software Environment

This manual has been prepared assuming that ALTIBASE HDB 6 will be used as the database server.

Organization

This manual is organized as follows:

- Chapter1: Installing ALTIBASE HDB
- Chapter2: Startup and Shutdown
- Chapter3: Working with ALTIBASE HDB
- Chapter4: DB Objects and Privileges
- Chapter5: Multilingual Features

  This chapter describes the multilingual features and related environment settings, and notes some relevant considerations.
- Chapter6: Database Replication

  This chapter explains in brief how to perform replication.

- Chapter7: Fail-Over
- Chapter8: Backup and Recovery
- Chapter9: Developing ALTIBASE HDB Applications

  This chapter introduces the various APIs (Application Programming Interfaces): Altibase CLI, ODBC, JDBC, C/C++ Precompiler, etc.

Documentation Conventions

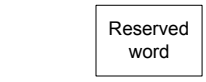
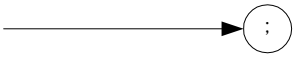
This section describes the conventions used in this manual. Understanding these conventions will make it easier to find information in this manual and other manuals in the series.

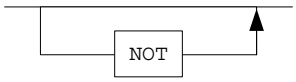
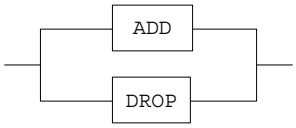
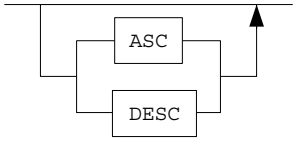
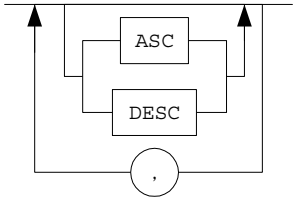
There are two sets of conventions:

- syntax diagrams
- sample code conventions

Syntax diagrams

This manual describes command syntax using diagrams composed of the following elements:

Elements	semantics
	The command starts. The syntax element which is not a complete command starts with an arrow.
	The command continues to the next line. The syntax element which is not a complete command terminates with this symbol.
	The command continues from the previous line. The syntax element which is a complete command starts with this symbol.
	End of the statement.
	Mandatory

Elements	semantics
	Optional
	Mandatory field with optional items. Only one field must be provided.
	Optional field with optional item
	Optional multiple fields are allowed. The comma must be in front of every repetition.

Sample Code Conventions

The code examples explain SQL, stored procedures, iSQL, and other command line syntax.

The following table describes the printing conventions used in the code examples.

Rules	Semantics	Example
[]	Indicates optional fields.	`VARCHAR [(size)] [[FIXED |] VARIABLE]`
{}	Indicates mandatory fields. Indicates to make sure to select at least one.	`{ ENABLE | DISABLE | COMPILE }`
|	Argument indicating optional or mandatory fields	`{ ENABLE | DISABLE | COMPILE }` `[ ENABLE | DISABLE | COMPILE ]`

Rules	Semantics	Example

.
.
.
	Repetition of the previous argument.Omit the example codes.	`iSQL> select e_lastname from employees;`
		`E_LASTNAME`
		`------------------------`
		`Moon`
		`Davenport`
		`Kobain`
		`.`
		`.`
		`.`
		`20 rows selected.`

Other symbols	Other symbols	`EXEC :p1 := 1;` `acc NUMBER(11,2);`
Italicized words	Indicates variable or value that must be provided by user.	`SELECT * FROM table_name;` `CONNECT` *`userID/password`*`;`
Lower case words	Program elements provided by the user such as table names, column names, file names, etc.	`SELECT e_lastname FROM employees;`
Upper case words	Elements provided by the system or keyword appeared in the syntax.	`DESC SYSTEM_.SYS_INDICES_;`

Related Documents

For more detailed information, please refer to the following documents:

- ALTIBASE HDB Installation Guide
- ALTIBASE HDB Administrator's Manual
- ALTIBASE HDB Replication Manual
- ALTIBASE HDB Precompiler User's Manual
- ALTIBASE HDB CLI User's Manual
- ALTIBASE HDB ODBC User' Manual
- ALTIBASE HDB Application Program Interface User's Manual
- ALTIBASE HDB iSQL User's Manual
- ALTIBASE HDB Utilities Manual
- ALTIBASE HDB Error Message Reference

On-line Manuals

Online versions of our manuals (PDF or HTML) are available from Altibase's Customer Support site (http://support.altibase.com/).

Altibase Welcomes Your Comments

Please let us know what you like or dislike about our manuals. To help us with future versions of our manuals, please tell us about any corrections or classifications that you would find useful.

Include the following information :

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

When you need an immediate assistance regarding technical issues, please contact Altibase's Customer Support site (http://support.altibase.com/).

Thank you. We appreciate your feedback and suggestions.

1 Installing ALTIBASE HDB

This chapter provides only a brief overview of the installation procedure. For complete instructions on how to install ALTIBASE HDB server, it is strongly recommended that you consult the *ALTIBASE HDB Installation Guide*.

This chapter contains the following sections:

1.1 The Installation Process

1.1.1 Download the Package Installer

Visit altibase.com and download the installer that is suitable for your operating system.

For information on the package installer files, please refer to Chapter 2 "Installing Products Using the ALTIBASE HDB Package Installer" of *Installation Guide*.

1.1.2 Run the Package Installer

For further information on how to execute the package installer, please refer to Chapter 2 "Installing Products Using the ALTIBASE HDB Package Installer" of *Installation Guide*.

When the package installer is executed, the following tasks are performed automatically:

1.1.2.1 Create the ALTIBASE_HOME Directory

This is the directory that contains the executable files, configuration files, and by default, the data and log files. The package installer gives you the opportunity to specify this directory.

1.1.2.2 Set the ALTIBASE HDB Property Values

The package installer suggests default property settings, and gives you the opportunity to change them as desired. These properties can be changed after installation is complete by modifying the altibase.properties file, which is located at ALTIBASE_HOME/conf/altibase.properties.

1.1.2.3 Create a Database Creation Script

The package installer can be used to create a script that you can later execute to create a database automatically, which greatly simplifies the database creation task.

1.1.3 Create a Database

After installation is complete, it will be necessary to create a database.

If a database was not created during the installation, you must create the database as follows:

- If you have set the properties for creating the database during the installation, you can create the database by executing $ALTIBASE_HOME/install/post_install.sh script.

  ```
  $ sh post_install.sh dbcreate
  ```

- If you have not set the properties during the installation, you can create the database by executing $ALTIBASE_HOME/bin/server script with arguments as follows:

  ```
  $ server create utf8 utf8
  ```

1.2 Special Considerations

1.2.1 Considerations when Installing ALTIBASE HDB in UNIX

1.2.1.1 Configuring Kernel Parameters

If system kernel parameters were not set during installation, please set the kernel parameters manually by referring to the following manuals:

- Installation Guide: Chapter 2 > Installing Altibase Products for Unix and Linux > Checking System Parameters
- Installation Guide: Appendix A > Setting Kernel Parameters for Different Operating Systems

To set system kernel parameters, log in as the root user.

1.2.2 Considerations when Installing ALTIBASE HDB in Windows

1.2.2.1 You Must Be Logged On as an Administrator

The ALTIBASE HDB Package Installer registers the ALTIBASE HDB ODBC driver, registers the ALTIBASE HDB server as a Windows service, and starts the service. Only administrators can register the ODBC driver, register the process as a Windows service, and start the service.

For further information on how to grant privileges to users, please refer to the following document:

- Installation Guide: Installing Altibase Products for Windows > Checking the Environment Before Installation

2 Startup and Shutdown

This chapter explains how to start up and shut down ALTIBASE HDB after it has been properly installed.

This chapter contains the following sections:

2.1 Startup

The ALTIBASE HDB server can be started up in one of two ways: either using a server script, or when a DBMS administrator logs in using the sys account, accesses the DBMS in sysdba administrator mode, and explicitly starts ALTIBASE HDB.

To explain the ALTIBASE HDB server startup process, first, the properties are read and system memory is checked, and then the ALTIBASE HDB system environment is initialized, system data are initialized, signal handling is initialized, the memory used for database space is initialized, the Query Processor is initialized, and finally, the threads are initialized. This completes ALTIBASE HDB server startup.

The command to start up ALTIBASE HDB can only be given using the Unix account with which ALTIBASE HDB was installed. The following shows how to start up a database using the iSQL utility, which ships with ALTIBASE HDB. For more information on the iSQL utility, please refer to the *ALTIBASE HDB iSQL User's Manual*.

```
$ isql -u sys -p manager -sysdba
----------------------------------------------------
 Altibase Client Query utility.
 Release Version 6.1.1.1
 Copyright 2000, ALTIBASE Corporation or its subsidiaries.
 All Rights Reserved.
----------------------------------------------------
ISQL_CONNECTION = UNIX, SERVER = 127.0.0.1, PORT_NO = 20300
iSQL(sysdba)>

iSQL(sysdba)> startup
Connecting to the DB server.... Connected.
TRANSITION TO PHASE : PROCESS
TRANSITION TO PHASE : CONTROL
TRANSITION TO PHASE : META
 [SM] Recovery Phase - 1 : Preparing Database
 : Dynamic Memory Version => Parallel Loading
 [SM] Recovery Phase - 2 : Loading Database
 [SM] Recovery Phase - 3 : Skipping Recovery & Starting Threads...
 Refining Disk Table
 [SM] Refine Memory Table :
..................................................
[SUCCESS]
[SM] Rebuilding Indices [Total Count:100]
..........................
[SUCCESS]
TRANSITION TO PHASE : SERVICE
 [CM] Listener started : TCP on port 20300
 [CM] Listener started : UNIX
[RP] Initialization : [PASS]
--- STARTUP Process SUCCESS ---
Command execute success.
```

The following shows how to start up a database using a server script.

```
$ server start
----------------------------------------------------
 Altibase Client Query utility.
```

```
 Release Version 6.1.1.1
 Copyright 2000, ALTIBASE Corporation or its subsidiaries.
 All Rights Reserved.
--------------------------------------------------------
ISQL_CONNECTION = UNIX, SERVER = 127.0.0.1, PORT_NO = 20300
[ERR-910FB : Connected to idle instance]
Connecting to the DB server... Connected.
TRANSITION TO PHASE : PROCESS
TRANSITION TO PHASE : CONTROL
TRANSITION TO PHASE : META
 [SM] Recovery Phase - 1 : Preparing Database
 : Dynamic Memory Version => Parallel Loading
 [SM] Recovery Phase - 2 : Loading Database
 [SM] Recovery Phase - 3 : Skipping Recovery & Starting Threads...
 Refining Disk Table
 [SM] Refine Memory Table :
..................................................... [SUCCESS]
 [SM] Rebuilding Indices [Total Count:100] ..............................
[SUCCESS]
TRANSITION TO PHASE : SERVICE
 [CM] Listener started : TCP on port 20300
 [CM] Listener started : UNIX
 [RP] Initialization : [PASS]
--- STARTUP Process SUCCESS ---
Command execute success.
```

2.2 Shutting Down ALTIBASE HDB

The ALTIBASE HDB server can be shut down either using a server script or when a user with DBMS administrator privileges uses iSQL to shut down ALTIBASE HDB as the sys user (with the -sysdba parameter). The "shutdown" command, which is the command that is used to shut down the server, has three mutually exclusive options. The way in which the server is shut down is different for each option. The ALTIBASE HDB shutdown command can only be given from the account used to install ALTIBASE HDB.

2.2.1 normal

In order for the server to shut down normally, the server must first wait until all clients have disconnected. If server shutdown is initiated while some tasks are still underway, the server waits for processes to terminate in the following order: first threads that sense client-server communication are shut down, followed by service threads, the Data Storage Manager, and finally the ALTIBASE HDB server process. At this time, the ALTIBASE HDB server has been completely shut down. When the server is shut down in this way, the following message is output.

```
iSQL(sysdba)> shutdown normal
Ok..Shutdown Proceeding....
TRANSITION TO PHASE : Shutdown Altibase
[RP] Finalization : PASS
shutdown normal success.
```

2.2.2 immediate

When the immediate shutdown option is used, before the server is shut down, connected sessions are forcibly disconnected, and then current transactions are forcibly rolled back.

The output for immediate shutdown is as follows:

```
iSQL(sysdba)> shutdown immediate
Ok..Shutdown Proceeding....
TRANSITION TO PHASE : Shutdown Altibase
 [RP] Finalization : PASS
shutdown immediate success.
```

The server can also be forcibly shut down using a server script command.

```
$ server stop
-------------------------------------------------------
 Altibase Client Query utility.
 Release Version 6.1.1.1
 Copyright 2000, ALTIBASE Corporation or its subsidiaries.
 All Rights Reserved.
-------------------------------------------------------
ISQL_CONNECTION = UNIX, SERVER = 127.0.0.1, PORT_NO = 20300
Ok..Shutdown Proceeding....
TRANSITION TO PHASE : Shutdown Altibase
 [RP] Finalization : PASS
shutdown immediate success.
```

2.2.3 abort

This option forces termination of an ALTIBASE HDB server with the system command 'kill -9'. When ALTIBASE HDB is shut down in this way, the database may not be closed properly, and thus database recovery will need to be performed when ALTIBASE HDB is restarted.

When the abort option is used, the following is output:

```
iSQL(sysdba)> shutdown abort
iSQL(sysdba)>
```

The server can also be forcibly shut down using a server script command.

```
$ server kill
-------------------------------------------------------
 Altibase Client Query utility.
 Release Version 6.1.1.1
 Copyright 2000, ALTIBASE Corporation or its subsidiaries.
 All Rights Reserved.
-------------------------------------------------------
ISQL_CONNECTION = UNIX, SERVER = 127.0.0.1, PORT_NO = 20300
$
```

3 Working with ALTIBASE HDB

This chapter contains the following sections:

3.1 Supported SQL Statements

Now that you have created a database and know how to start it up and shut it down, you will of course want to execute some SQL statements, starting with DDL statements for creating other users and database objects such as tables, and then DML statements to populate the tables with data and perform similar actions. For a description of the various kinds of available database objects, please refer to Chapter4: DB Objects and Privileges.

ALTIBASE HDB Server supports the complete ANSI 92 SQL standard, and additionally provides some extended functionality. For a complete description of all SQL statements that are supported by ALTIBASE HDB Server, please refer to the *SQL Reference*.

3.2 How to Execute SQL Statements

3.2.1 Executing SQL Statements using the iSQL Utility

The most straightforward way to execute SQL statements is to use the iSQL utility, which is a command-line interface that ships with ALTIBASE HDB. For complete information on how to use the iSQL utility, please refer to the *iSQL User's Manual*.

3.2.2 Executing SQL Statements using Custom-Authored Client Applications

Additionally, SQL statements can also be executed using custom-authored client applications. The point of entry for complete information on authoring applications for use with ALTIBASE HDB is Chapter9: Developing ALTIBASE HDB Applications.

3.3 The Sample Schema

The ALTIBASE HDB server package includes a script, which, when executed, creates a sample schema that includes a series of database tables and other objects, and populates the tables with sample data. Simply execute the script, which is located at ALTIBASE_HOME/sample/APRE/schema/schema.sql, and the sample schema is created for you.

Many of the examples in the product documentation are based on this sample schema. If you wish to follow along with the examples, or simply need some sample data on which to execute SQL statements for practice, it is recommended that you execute the above script to create the sample schema.

For a complete description of all of the objects and data in the sample schema, please refer to the *ALTIBASE HDB General Reference*.

4 DB Objects and Privileges

In this chapter, schema objects and non-schema objects will be classified, and the database objects in each category will be explained.

This chapter contains the following sections:

4.1 Database Objects: An Overview

Database objects can be divided into schema objects, which belong to certain schema, and non-schema objects, which do not have any relationship with particular schema. In this chapter, schema objects and non-schema objects will be classified, and the database objects in each category will be explained.

4.1.1 Schema Objects

Schemas are logical collections of data and objects. Relational schemas are grouped by database user ID and include tables, views, and other objects. A user owns a schema and manages it using SQL statements. The objects included in schemas are called schema objects. ALTIBASE HDB supports the following schema objects:

4.1.1.1 Tables

A table is the basic unit for storing data, and is a set of records consisting of columns. ALTIBASE HDB tables are divided into memory tables and disk tables depending on how the data are stored, and are also divided, based on who creates them, into system tables, which are created and managed by the system, and user tables, which are created by general users.

System tables are also called the "data dictionary". For detailed information about the data dictionary provided with ALTIBASE HDB, as well as data dictionary management issues, please refer to the Data Dictionary in Chapter 2 of the *General Reference*.

Additionally, replication target tables and large volume tables also have special issues related to their management.

For more detailed information than can be found here, the portion of the *Administrator's Manual* pertaining to database objects describes in great detail how to manage them.

4.1.1.2 Partitioned Table

When a table is partitioned, the table is called a partitioned table. A partitioned table is a large table that has been divided into several partitions based on the partitioning conditions (range, list and hash).

For more information, please refer to the portion of the *Administrator's Manual* pertaining to partitioned tables.

4.1.1.3 Partitioned Index

Indexes are categorized as partitioned indexes or non-partitioned indexes based on whether or not they are partitioned. Non-partitioned indexes are indexes that have not been partitioned, while partitioned indexes (like partitioned tables) are large indexes that have been divided into several indexes based on some partitioning conditions.

For more information, please refer to the portion of the *Administrator's Manual* pertaining to partitioned indexes.

4.1.1.4 Temporary Tables

Temporary tables are used to temporarily hold data for the duration of a session or transaction. The use of temporary tables can enhance the performance speed of complex queries.

Temporary tables can only be created on volatile tablespace.

4.1.1.5 Queue Tables

ALTIBASE HDB supports asynchronous data communication between user applications and the database using message queuing functionality. Queue tables are manipulated using DML and DDL statements, just like other database tables. For more information on the concepts and functionality of queue tables, please refer to the portion of the *ALTIBASE HDB Administrator's Manual* pertaining to database objects.

4.1.1.6 Constraints

Constraints serve to restrict data manipulation in order to maintain data consistency when data are inserted into tables, or when existing data in tables are changed. Depending on the target of the constraints, constraints are divided into column constraints and table constraints. ALTIBASE HDB supports the following kinds of constraints.

- NULL/NOT NULL Constraints
- CHECK Constraints
- Unique Key Constraints
- Primary Key Constraints
- Foreign Key Constraints
- TIMESTAMP Constraints

For more detailed information than can be found here, please refer to the portion of the *Administrator's Manual* pertaining to constraints.

4.1.1.7 Indexes

Indexes are elements that allow records within tables to be accessed more quickly. Indexes are created within tables and increase the performance with which DML statements are processed.

For more information, please refer to the portion of the *Administrator's Manual* dealing with indexes.

4.1.1.8 Views

A view does not contain actual data, but is a logical table created on the basis of one or more tables or views.

For more information, please refer to the portion of the *Administrator's Manual* pertaining to views.

4.1.1.9 Materialized View

A materialized view is a database object that stores query results as data. Data can be based on more than one table, a view, or another materialized view. For more information, please refer to the portion of the *Administrator's Manual* pertaining to materialized views.

4.1.1.10 Sequences

ALTIBASE HDB provides sequences for generating unique keys. For more information, please refer to the portion of the *Administrator's Manual* pertaining to sequences.

4.1.1.11 Synonyms

Synonyms are provided as aliases for tables, sequences, views, stored procedures and stored functions so that they can be used without being accessed directly by the object name. For more information, please refer to the portion of the *Administrator's Manual* pertaining to synonyms.

4.1.1.12 Stored Procedures and Functions

A stored procedure is a kind of database object in which all kinds of operations, such as SQL statements, stream control statements, assignment statements, and error handling routines, are programmatically combined into a single module that is permanently stored in the database, after which all of the operations stored therein can be executed merely by calling the stored procedure using its name.

For more information, please refer to the portions of the *Administrator's Manual* pertaining to stored procedures and stored functions. Additionally, for detailed information about the special features of stored procedures as provided with ALTIBASE HDB, as well as how to manage them, please refer to the *Stored Procedures Manual*.

4.1.1.13 Type Sets

A type set is a database object which allows a number of user-defined data types that are used by stored procedures and stored functions to be grouped together in one place for convenient management. For more about type sets, please refer to the *Stored Procedures Manual*.

4.1.1.14 Database Triggers

A trigger is a kind of stored procedure that is called automatically by the system when data in a table are inserted, deleted, or updated, thus allowing a specific set of tasks to be automatically performed. By defining constraints and triggers for tables, the user can maintain data consistency.

For more information, please refer to the portion of the *Administrator's Manual* pertaining to triggers.

4.1.1.15 Database Link

Database Link allows database servers that are geographically distributed but connected via a network to be linked together to combine the data thereon and output a single result.

4.1.1.16 External Procedures and Functions

External procedures or external function objects are database objects that correspond to user-defined C/C++ functions on a one-to-one basis. User-defined functions are executed through external procedure objects or external function objects. Depending on whether or not they return values differentiates external procedures from external functions.

For further information, please refer to *C/C++ External Procedures Manual*.

4.1.1.17 Library

The dynamic library file containing user-defined C/C++ functions to be used with external procedures must be identifiable by the ALTIBASE HDB server. For this purpose, ALTIBASE HDB provides the library object which is a database object that corresponds to the dynamic library file on a one-to-one basis.

For further information, please refer to *C/C++ External Procedures Manual*.

4.1.2 Non-schema Objects

Objects that do not belong to any particular schema, and are thus managed at the level of the entire database, are called non-schema objects. ALTIBASE HDB provides the following non-schema objects:

4.1.2.1 Directories

Stored procedures are able to control files, which allows them to read from and write to text files in the file system managed by the operating system. Thanks to this functionality, the user can perform various kinds of tasks using stored procedures such as leaving messages in files, reporting the results of files or reading data from files for insertion into tables. The directory object is used to manage information about the directories accessed by stored procedures.

Please refer to the *SQL Reference* for a detailed description of the directory object.

Please refer to the *Stored Procedures Manual* for a detailed explanation of how to handle files using stored procedures.

4.1.2.2 Replication

Replication can be thought of as a kind of object that allows information to be automatically sent from a local server to a remote server so that the data in tables on multiple servers can be kept consistent.

For more details on replication, please refer to the *Replication Manual*.

4.1.2.3 Tablespaces

The tablespace is the largest logical data storage unit. A database consists of and manages multiple tablespaces. ALTIBASE HDB automatically creates a system tablespace when a database is created, and the user can create user-defined tablespaces as desired.

ALTIBASE HDB supports 3 types of tablespaces: disk tablespace, which resides on disk, memory

tablespace, which resides in memory, and volatile tablespace, which also resides in memory but differs from memory tablespace in that logging is not performed.

For more information on tablespace management, please refer to Chapter 6: Tablespace" in the *Administrator's Manual*.

4.1.2.4 Users

A user is the owner of a schema, and is associated with a user account that is required in order to access a database. Users are created by the system, and are categorized as either system users, who manage the entire system, or general users.

Users must have been granted appropriate privileges in order to access and manage the database.

For more information, please refer to "Chapter 5: Objects and Privileges" in the *Administrator's Manual*.

4.1.2.5 Jobs

A JOB is the addition of an execution schedule to a stored procedure. The stored procedure to be executed, the point in time of execution, the interval after which it is to be executed and etc. can be set when creating the JOB object. For the created JOB to automatically run, the value of the JOB_SCHEDULER_ENABLE property must be set to 1.

The creation, alteration and deletion of the JOB, and the management of the job scheduler is only enabled for the SYS user.

For more information, please refer to "Chapter 5: Objects and Privileges" in the *Administrator's Manual*.

4.2 Privileges

Users must have appropriate privileges in order to access database objects and data. This chapter describes the privileges pertaining to users and objects and how to manage them.

4.2.1 Managing Privileges

ALTIBASE HDB supports system privileges, object privileges and roles.

4.2.1.1 System Privileges

System access privileges are usually managed by the DBA (Database Administrator). Users with system privileges can execute individual tasks and manage all objects in all schemas.

4.2.1.2 Object Privileges

The object owner manages object privileges, which are the right to access and manipulate objects.

For a complete list of the privileges supported in ALTIBASE HDB, please refer to the portion of the *ALTIBASE HDB Administrator's Manual* dealing with privilege management, and for more detailed information about statements for granting and revoking privileges, please refer to the *ALTIBASE HDB SQL Reference*.

4.2.1.3 Roles

A role is a group of privileges; by using roles, you can easily grant multiple privileges to users. For further information on roles and their restrictions, please refer to the *SQL Reference*.

4.2.2 Granting Privileges

When a database is in an initialized state immediately after it has been created, the SYSTEM_ and SYS users already exist, have all DBA privileges, and can grant privileges to normal users.

When a normal user is created using the CREATE USER statement, the system automatically grants the user the minimum privileges necessary to access the database, such as the authority to execute CREATE SESSION and CREATE TABLE statements. Other privileges must be explicitly granted by the DBA.

For more detailed information on how to grant and manage privileges, please refer to the relevant portion of the *Administrator's Manual* and to the *SQL Reference*.

4.2.3 Revoking Privileges

Privileges granted to users other than the SYSTEM_ and SYS users can be explicitly revoked using the REVOKE statement.

Even the privileges that are automatically granted by the system when a user is created using the CREATE USER statement can be revoked if necessary.

However, the privileges of the SYSTEM_ and SYS users cannot be revoked.

5 Multilingual Features

This chapter describes the multilingual structure of ALTIBASE HDB, as well as environment settings and other points to consider when using ALTIBASE HDB in a multilingual environment.

This chapter contains the following sections:

5.1 Multilingual Support Overview

5.1.1 Concept

Multilingual support means that the database is capable of storing and processing character sets used in different countries. In other words, a single database can handle clients that use different languages (for example, Korean, Chinese, and Japanese).

5.1.2 Related Terminology

- Character Set

  A character set is a particular group of characters that are associated with respective numeric values. The following table shows how an individual character is associated with a different numeric value depending on whether it is encoded using the UTF-8, UTF-16 BE or UTF-16 LE character set.

Character	UTF-8	UTF-16 BE	UTF-16 LE
A	41	00 41	41 00
Ő	C3 B6	00 F6	F6 00

- NLS (National Language Support)

  This allows the database to be used in a particular language environment. If NLS is appropriately set, the user can read and write data to and from the database using the character set specified by the user's application.

5.1.3 Multilingual Support

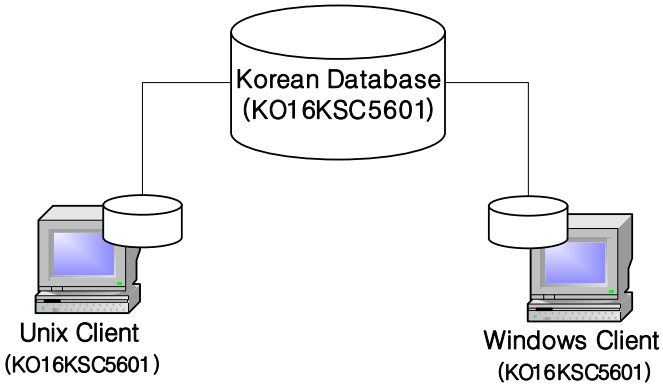
Multi-language support consists of performing conversions between the character sets used by the database and the client application, respectively. From the aspect of multilingual support, the server and client can have one of four relationships therebetween, which are explained individually below:

- The database and the client use the same character set.
- The database and the client use different character sets.
- The database and multiple clients use different character sets.
- Unicode data types are supported by both the database and the client.

5.1.3.1 The database and the client use the same character set.

The character set used by the database is the same as the character set used by the client.

Figure 5-1 A Database and a Client with the Same Character Set



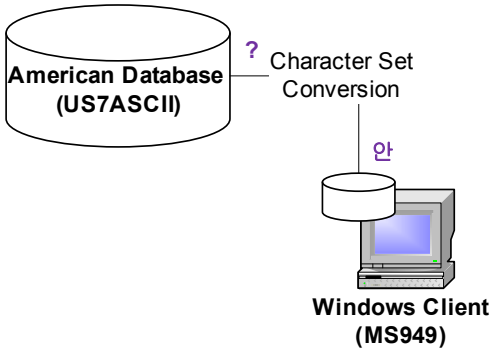
If both the database and the client use KSC5601 as the character set, as shown in *Figure 6-1*, character set conversion need not be performed.

5.1.3.2 The database and the client use different character sets.

If the character set used by the database is different from the character set used by the client, character set conversion occurs.

This can sometimes lead to data loss, as shown in *Figure 6-2*.

Figure 5-2 A Database and a Client with Different Character Sets



To prevent data loss caused by character set conversion, it is recommended that the character set used on the database be a superset of the character set used by the client.

Thus, to prevent data loss when character conversion is performed as seen in the picture above, the character set used by the database should be MS949 or UTF8, which is a superset of MS949.

5.1.3.3 The database and multiple clients use different character sets.

If multiple client applications use different character sets, specifying that the server uses a character set that encompasses all of the character sets used by the clients can prevent data loss attributable to character set conversion.

Figure 5-3 A Database and Multiple Clients with Different Character Sets

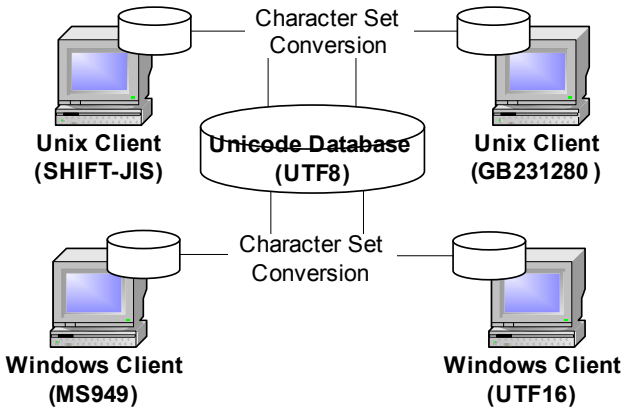


Figure 5-3 illustrates a system in which clients that are connected to the same database use Japanese, Chinese and Korean. In order to prevent data loss caused by character set conversion, the database character set should be set to UTF8, which supports the languages used by all of these clients.

5.1.3.4 Unicode data types are supported by both the database and the client.

If the database and client application use NCHAR or NVARCHAR, both of which support Unicode data, multiple languages are supported, regardless of which character set each of them is using.

5.2 Character Set Classification for Multilingual Support

5.2.1 Database Character Set

The “database character set” is the character set with which data are saved in the database.

Any character set that encompasses (completely includes) the ASCII character set as per the SQL standard can be used as the database character set. Thus, UTF-16 can't be used as the database character set because UTF-16 doesn't encompass the ASCII character set.

5.2.1.1 How to Specify the Database Character Set

The database character set can be specified using the CREATE DATABASE statement when a database is created.

5.2.1.2 Supported Database Character Sets

ALTIBAS supports the use of the following 8 database character sets, all of which support ASCII:

- US7ASCII
- KO16KSC5601
- MS949
- BIG5
- GB231280
- MS936 (Identical to the GBK, ZHS16GBK, CP936 character sets of other vendors)
- UTF8
- SHIFTJIS
- EUCJP

5.2.2 National Character Set

The national character set is used to store NCHAR and NVARCHAR data types, and can be used to store text in Unicode.

5.2.2.1 How to Specify the National Character Set

The national character set of the database is specified using the CREATE DATABASE statement when a database is created.

5.2.2.2 Supported National Character Sets

ALTIBASE HDB supports the following two national character sets:

- UTF8
- UTF16 (Big Endian)

5.2.3 Client Character Set

The client character set is the character set used to display data to the client.

Data sent from the server are converted to, and displayed in, the character set specified by respective clients.

5.2.3.1 How to Specify the Client Character Set

The client character set can be specified using ALTIBASE_NLS_USE on the client.

5.2.3.2 Supported Client Character Sets

- US7ASCII (Default)
- KO16KSC5601
- MS949
- BIG5
- GB231280
- MS936
- UTF8
- UTF16 (Big Endian)
- SHIFTJIS
- EUCJP

5.3 Using Unicode

in a Multilingual Database

5.3.1 The Unicode Concept

Unicode is an internationally encoded character set that enables information to be stored in any language using a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language.

Therefore, Unicode is useful when it is necessary to save data in several languages simultaneously.

5.3.2 Unicode Encoding

Unicode encoding is the way Unicode data are represented so that they can be stored on a computer. Unicode, which means a system or collection of characters, requires an encoding system such as UTF-8 or UTF-16 in order to encode data.

5.3.3 Storing Unicode Characters

Unicode characters can be stored in a database in two ways:

- When the database is created, it can be designated as one in which character data are stored as Unicode data.
- NCHAR or NVARCHAR columns can be used to store Unicode characters.

Please note that, if the database character set is UTF8 and the national character set is UTF16, Unicode characters may be stored in two different ways in the same database.

5.3.4 A Unicode Database

If the database character set is set to UTF8 at the time that the database is created to thus create a Unicode database, then Unicode data can be saved in CHAR and VARCHAR type columns.

5.3.4.1 Supported Character Set

- UTF8

5.3.4.2 When is a Unicode database needed?

- When SQL statements or stored procedures include Unicode data.
- When you are not sure whether multilingual data will be inserted into the database, or what column they will be inserted into.

5.3.5 Unicode Datatypes

Even if a character set other than UTF8 was specified at the time a database was created, it is still possible to store Unicode characters using the NCHAR or NVARCHAR data type.

5.3.5.1 Supported Character Sets

- UTF8
- UTF16

5.3.5.2 When are Unicode data types necessary?

- When columns for storing multilingual data are needed in a non-Unicode database.
- When most of the data to be saved in a given column are in the same language, but some of the data to be saved in that column are in some other language(s).
- When saving data from a wchar_t (16-bit) buffer of a Windows client application.

5.4 Making Environment Settings for a Multilingual Database

In order to establish a database that supports multiple languages, settings must be made as follows:

1. When creating a database, consider which character set is the most widely used by clients, and specify that character set for the server.
2. Set NLS appropriately for the client character set.
3. Set other environment variables and properties.

5.4.1 Setting Environment Variables

Set the following environment variables on the clients:

- ALTIBASE_NLS_USE
- ALTIBASE_NLS_NCHAR_LITERAL_REPLACE

5.4.1.1 ALTIBASE_NLS_USE

Any of the following character sets may be used on the clients. Data sent from the server are converted to, and displayed in, the character set specified by each of the clients.

- US7ASCII (Default)
- KO16KSC5601
- MS949
- BIG5
- GB231280
- MS936
- UTF8
- SHIFTJIS
- EUCJP

5.4.1.2 ALTIBASE_NLS_NCHAR_LITERAL_REPLACE

If this is set to 1(TRUE), the client does not convert strings that are preceded by the "N" character to the database character set. Rather, it sends them to the server without change, and the server converts them to the national character set. The default is 0 (FALSE).

Queries used by client applications are usually converted to the database character set and then sent to the server. Under this scheme, for a database that uses the US7ASCII character set, data that fall out of the range of the US7ASCII character set can't be inserted into that database, even if an

NCHAR column is created for that purpose.

For example, if the client character set is KO16KSC5601 and the database character set is US7ASCII, data are converted from the client character set to the database character set when an INSERT statement is executed. In this case, as can be seen in the following example, because it can't be converted to US7ASCII, the replacement character '?' is stored in the table.

```
iSQL> CREATE TABLE t1 ( i1 NVARCHAR(10) );
Create success.

iSQL> INSERT INTO t1 VALUES ('안');
1 row inserted.

iSQL> SELECT * FROM t1;
I1
--------------------
??
```

Therefore, a method of saving data that does not fall within the range of the database character set in an NCHAR column is needed. In one such method, seen below, an environment variable setting is made and data are inserted using the NCHAR literal:

```
$ export ALTIBASE_NLS_NCHAR_LITERAL_REPLACE=1
...

iSQL> CREATE TABLE t1 ( i1 NVARCHAR(10) );
Create success.

iSQL> INSERT INTO t1 VALUES ( N'안' );
1 row inserted.

iSQL> SELECT * FROM t1;
I1
--------------------
안
```

As can be seen above, If ALTIBASE_NLS_NCHAR_LITERAL_REPLACE is set to 1(TRUE) and data are inserted, the client does not convert strings that are preceded by the "N" character to the database character set. Instead, these strings are sent to the server without change, where they are converted to the national character set.

5.4.2 Example

The following explains the process of setting up an environment in which the default database character set is KSC5601 and UTF16 is used as the national character set.

5.4.2.1 Database Creation

```
iSQL(sysdba)> CREATE DATABASE mydb INITSIZE=10m NOARCHIVELOG CHARACTER SET
KSC5601 NATIONAL CHARACTER SET UTF16;

DB Info (Page Size = 32768)
       (Page Count = 257)
       (Total DB Size = 8421376)
       (DB File Size = 1073741824)
         Creating MMDB FILES [SUCCESS]
         Creating Catalog Tables [SUCCESS]
```

```
        Creating DRDB FILES [SUCCESS]
  [SM] Rebuilding Indices [Total Count:0] [SUCCESS]
DB Writing Completed. All Done.
Create success.
```

5.4.2.2 Making Environment Settings on the Client

To use KSC5601 on the client, set the environment variable as follows:

```
$ export ALTIBASE_NLS_USE=KSC5601
```

To use ASCII on the client, set the environment variable as follows:

```
$ export ALTIBASE_NLS_USE=ASCII
```

5.4.2.3 Setting Other Environment Variables and Properties

Set the following environment variable and property appropriately for the usage environment.

- Environment Variable

  ALTIBASE_NLS_NCHAR_LITERAL_REPLACE

- Property

  NLS_COMP

  or

  NLS_NCHAR_CONV_EXCP

5.5 Considerations when Choosing a Database Character Set

When choosing a database character set, please give careful consideration to any issues that could arise, including those associated with identifiers as well as data loss and inadvertent conversion that may occur when data are converted.

5.5.1 Scope of Usage

5.5.1.1 Identifiers

Column names, schema objects and comments are saved in the database using the database character set, however, other identifiers can only be stored using the US7ASCII character set.

The following table shows which character sets can be used for each kind of identifier.

Table 5-1 Character Sets that Can Be Used for each Identifier

Identifier Name	Available Character Set
Column Name	Database Character Set
Schema Object	Database Character Set
Annotation	Database Character Set
Database Link Name	Database Character Set
Database Name	US7ASCII
File Name(Such as Data and Log Files)	US7ASCII
Directory Name	US7ASCII
Keyword	US7ASCII
Tablespace Name	US7ASCII

5.5.1.2 Stored SQL Statements

SQL statements that are stored in meta tables, such as those belonging to triggers and stored procedures, are stored using the database character set.

5.5.2 One Restriction

5.5.2.1 Replication

Replication cannot be performed between two databases that use different character sets.

5.5.3 Effects of Character Set Conversion

If the database character set is different from the character set used by the client, character set conversion will occur. The possibility of data loss is not the only negative consequence; performance may also suffer.

5.5.3.1 Data Loss

When data are converted from a character set that can represent a wide range of characters to another with a narrower range, data loss can result.

Any characters that cannot be represented using the destination character set will be converted to a replacement character. In US7ASCII, the replacement character is the question mark ('?').

5.5.3.2 Conversion Overhead

If all clients use the same character set, and the same character set is specified when a database is created, no character conversion will occur.

However, if different character sets are in use on each client, and the database character set is a superset of the character sets used by the clients, character conversion will occur.

6 Database Replication

Replication is the operation of copying and maintaining database objects in multiple databases that make up a distributed database system. ALTIBASE HDB provides transaction log-based replication, so that when a database server experiences an unexpected outage, service can continue to be provided without any interruption. This chapter gives an overview of how to perform replication and broadly explains the related concepts.

This chapter contains the following sections:

6.1 Introduction to Replication

The Altibase database replication function maintains an up-to-date backup of the database on an active server, and in the event that the server is unexpectedly terminated, immediately resumes service again from an identical database on an alternative server, so as to realize an operating environment in which uninterruptible service is provided.

In this chapter, an explanation will first be given of how databases are replicated in ALTIBASE HDB, followed by instructions to help you replicate your databases properly. Please refer to the *ALTIBASE HDB Replication Manual* for more detailed information.

6.2 How Databases Are Replicated in ALTIBASE HDB

6.2.1 Establishing a Replication Environment

In order to make use of the replication functionality, first the tables containing the data to be replicated are defined and a schema comprising the remote server to be replicated, the replication name, the primary key, the port number etc. is set, and a replication connection is established between the local server and the remote server.

Then, replication of the data on the remote server can begin.

Bidirectional replication, in which replication is also initiated on the remote server, is also possible.

6.3 How to Replicate a Database

When database replication is performed in ALTIBASE HDB, the local server sends database changes that have occurred in the system to the remote server, and the remote server makes corresponding changes in its own database.

The local server and the remote server start threads dedicated to the task of replication. These threads are distinct from the database service threads. The replication Sender thread on the local server transmits the database changes, and the replication Receiver thread on the remote server receives the information about the data changes and implements them in its database.

Additionally, the replication Sender and Receiver threads automatically detect whether the corresponding servers were shut down normally or abnormally and take appropriate action.

6.3.1 Creating Replication Objects

Replication to synchronize a local server with a remote server is defined as follows:

```
CREATE [LAZY|EAGER] REPLICATION
  replication_name [AS MASTER|AS SLAVE]
  WITH 'remote_host_ip', remote_host_port_no
  FROM user_name.table_name TO user_name.table_name,
  FROM user_name.table_name TO user_name.table_name,
  …
  FROM user_name.table_name TO user_name.table_name;
```

6.3.2 Starting Replication

Replication is started in one of these ways:

```
ALTER REPLICATION replication_name SYNC [PARALLEL parallel_factor];
ALTER REPLICATION replication_name SYNC ONLY
  [PARALLEL parallel_factor];
ALTER REPLICATION replication_name START;
ALTER REPLICATION replication_name QUICKSTART;
```

6.3.3 Stopping Replication

Replication is stopped in this way:

```
ALTER REPLICATION replication_name STOP;
```

6.3.4 Resetting Replication

This is how replication information is reset. Replication must be stopped before this is done.

```
ALTER REPLICATION replication_name RESET;
```

6.3.5 Dropping Tables

This is how tables are dropped (deregistered) from a replication object. Replication must be stopped before this is done.

```
ALTER REPLICATION replication_name STOP;
ALTER REPLICATION replication_name DROP TABLE
  FROM user_name.table_name
  TO user_name.table_name;
```

6.3.6 Adding Tables

This is how tables are added to (registered with) a replication object. Replication must be stopped before this is done.

```
ALTER REPLICATION replication_name STOP;
ALTER REPLICATION replication_name ADD TABLE
  FROM user_name.table_name
  TO user_name.table_name;
```

6.3.7 Dropping a Replication Object

This is how a replication object is dropped. If replication has been started, it must first be stopped before the replication object can be dropped.

```
ALTER REPLICATION replication_name STOP;
DROP REPLICATION replication_name;
```

6.4 Executing DDL Statements in a Replication Environment

If the REPLICATION_DDL_ENABLE property is set to 1 on a replication server, the following DDL statements can be executed:

- ALTER TABLE *table_name* ADD COLUMN
- ALTER TABLE *table_name* DROP COLUMN
- ALTER TABLE *table_name* ALTER COLUMN *column_name* SET DEFAULT
- ALTER TABLE *table_name* ALTER COLUMN *column_name* DROP DEFAULT
- ALTER TABLE *table_name* TRUNCATE PARTITIONTRUNCATE TABLE
- CREATE INDEX
- DROP INDEX

Please refer to the *Replication Manual* for a complete list of the DDL statements that may be executed, as well as more information about restrictions pertaining to replication.

However, depending on the task at hand, DDL statements that are not normally permitted in a replication environment may be executed as long as replication is first paused, or replication definitions are first dropped. Additionally, DDL statements cannot be executed on table objects that are replication targets.

To execute such a DDL statement, replication must first be stopped on both servers, and the table in question must be dropped from the replication definition, after which DDL statements can be executed on each server. Finally, once the DDL statements have been executed successfully, the relevant tables are re-registered in the replication definitions, and replication is resumed.

7 Fail-Over

Fail-Over functionality is provided so that, when a fault occurs in a database system that is actively providing service, it can be overcome and service can continue to be provided, as though no fault had occurred. This chapter will explain how to use the Fail-Over functionality that is provided with ALTIBASE HDB.

This chapter contains the following sections:

7.1 About Fail-Over

7.1.1 The Fail-Over Concept

"Fail-Over" means the ability to overcome a fault that occurs in a database system that is actively providing service and continue to provide service despite the fault. The kinds of faults that can occur include: the case in which the hardware that the DBMS is operating experiences a fault, the network via which the server is connected experiences an outage, and the case in which the DBMS software encounters an error and shuts down abnormally. When one of these kinds of faults occurs (regardless of which kind it is) Fail-Over, due to its ability to connect to another DBMS server, enables service to be continuously provided without client applications ever being aware that a fault occurred.

One of the following two kinds of Fail-Over is performed, depending on the time point at which the fault is discovered:

- CTF (Connection Time Fail-Over)
- STF (Service Time Fail-Over)

With CTF, the fault is discovered at the time of connection to the DBMS, and connection is made to another DBMS at an available node, instead of the DBMS in which the fault occurred, so that service can be continuously provided.

With STF, because a fault occurs while service is being provided after successful connection to the DBMS, reconnection is made to a DBMS on another available node and session properties are restored so that the business logic of the user's application can continue to be used. That is to say, tasks currently being executed on the DBMS in which the fault occurred may need to be executed again.

When this kind of Fail-Over is conducted, in order to be confident in the results of a task, the databases on the DBMS in which the fault occurred and the DBMS that is available for service must be guaranteed to be in exactly the same state and contain exactly the same data.

In order to guarantee that the databases match, ALTIBASE HDB copies the database using Off-Line Replication. In Off-Line Replication, the stand-by server reads the logs from the active server so that it can harmonize its database with that on the active server.

Because one of the characteristics of replication is that the databases might not be in exactly the same state, we recommend that the Fail-Over Callback function be used to confirm that the databases match. Fail-Over Callback is explained in detail in the next chapter.

Fail-Over settings of ALTIBASE HDB include a Fail-Over property which is set to true to specify that Fail-Over is to be executed. Additionally, the Fail-Over Callback function can be used to check whether the databases match before Fail-Over is executed.

The three kinds of Fail-Over-related tasks that must be executed by the client application are summarized as follows:

- the Fail-Over connection property must be set to true
- the Fail-Over Callback function must be registered
- additional tasks may be necessary depending on the result of callback

For more detailed information, please refer to the *Replication Manual*.

7.2 How to Use Fail-Over

7.2.1 Setting the Fail-Over Connection Property

If the Fail-Over connection property has been set, when ALTIBASE HDB senses the occurrence of a fault, it conducts internal Fail-Over tasks as specified by the connection property.

There are two ways to show the property values:

- by viewing the Connection Property string used by the API's "Connect" function
- by viewing the ALTIBASE HDB properties files:

  the altibase_cli.ini file

  the odbc.ini file (WinODBC)

For more details about how to set this property, please refer to the *Replication Manual*.

7.2.2 Checking Whether Fail-Over Has Succeeded

In the case of CTF (Connection Time Fail-Over), attempting to connect to the database makes it immediately obvious whether Fail-Over was successful. In contrast, in the case of STF (Service Time Fail-Over), whether Fail-Over was successful is determined by checking for exceptions and errors.

For example, in the case of JDBC, when a SQLException is caught, the value of SQLStates.status is checked using the SQLException's getSQLState() method, and if this value is found to be ES_08FO01, then it is known that Fail-Over succeeded.

In the case of CLI and ODBC, if the result of a SQLPrepare, SQLExecute, or SQLFetch statement or the like is an error rather than SQL_SUCCESS, a statement handle is returned in response to SQLGetDiagRec, and if the result of the call to SQLGetDiagRec is ALTIBASE_FAILOVER_SUCCESS, then it is confirmed that STF (Service Time Fail-Over) succeeded.

When using embedded SQL, after executing an EXEC SQL statement, the value of the return code "sqlca.sqlcode" is checked, and if it is ALTIBASE_FAILOVER_SUCCESS (rather than SQL_SUCCESS), then it is confirmed that STF (Service Time Fail-Over) succeeded.

For more detailed information on these settings, please refer to the *Replication Manual*.

7.2.3 How to Write a Fail-Over Callback Function

The way to write a Fail-Over Callback function differs depending on the form of the client application, but the basic structure is usually the same, and consists of the following:

- defining Fail-Over-related data structures
- writing the body of Fail-Over Callback functions that will be called when Fail-Over-related events occur
- checking whether Fail-Over has succeeded

The Fail-Over event is either defined in the data structure definition or included in a defined interface (header file). The callback function body must include Fail-Over events, that is, tasks that must be conducted when Fail-Over starts or finishes, for example, code that checks whether databases match. If Fail-Over completes successfully, and the callback function also executes successfully, with the result that the service that was suspended by the fault can be used again, then Fail-0ver is considered to have been successful.

For specific information on how to write such functions in various client application environments, please refer to the *Replication Manual*.

8 Backup and Recovery

ALTIBASE HDB data can be lost due to unforeseen circumstances, such as in the event of system failure or the loss of, or damage to, a disk or data file. This chapter describes backup and recovery features of ALTIBASE HDB for use in preparing for such incidents.

This chapter contains the following sections:

8.1 ALTIBASE HDB Backup Policy

ALTIBASE HDB normally supports both logical backup and physical backup. Logical backup is for creating copies of database objects and saving data in text file format. The ALTIBASE HDB utilities “aexport” and “iLoader” can be used to perform logical backup. Logical backup does not support recovery up until the point in time at which the error occurred.

The logical backup procedure is as follows:

1. Export the database objects to text files using aexport or iLoader while ALTIBASE HDB is running.

2. Store the backup text files to disk or tape.

Physical backup means copying physical data files to disk or tape. ALTIBASE HDB supports both offline (“cold”) backup and online (“hot”) backup.

Performing offline backup consists of shutting down the database normally and backing up all files required by the database. The offline backup procedure is as follows:

1. Shut down the database normally.

2. Back up the log anchor file, online log files, and database files.

Online backup can be used when the database is in archive log mode. Online backup can be conducted while the database is providing service, but it is recommended that online backup be performed during periods when low usage is anticipated. If online backup is conducted during periods of high use, excessive logs can be generated.

The online backup procedure is as follows:

1. Back up memory tablespaces and log anchor files.

2. Back up disk tablespaces.

3. Back up archive log files.

8.2 ALTIBASE HDB Recovery Policy

ALTIBASE HDB provides the following recovery methods:

- logical backup recovery
- restart recovery
- media recovery

Logical backup recovery means recovery from backup text files using the iLoader utility. Restart recovery is a simple recovery method that is automatically initiated when an ALTIBASE HDB server is restarted. Restart recovery is conducted after a database server has been abnormally terminated.

Media recovery uses database files, log anchor files, and archive log files that were created based on the backup policy to recover database files either to the most recent backup or to a specific moment in time ("point-in-time recovery"). Depending on the kind of media error and the recovery procedure, the database may be recovered using either complete recovery or incomplete recovery, as appropriate. For more information about backup and recovery, please refer to the portion of the *ALTIBASE HDB Administrator's Manual* pertaining to backup and recovery.

9 Developing ALTIBASE HDB Applications

This chapter will provide an overview of the process of authoring client applications that access ALTIBASE HDB.

This chapter contains the following sections:

9.1 Writing Client Application Programs

Developing ALTIBASE HDB applications using the ALTIBASE HDB application program interface (API) for Altibase CLI, JDBC, ODBC, the C/C++ precompiler and the like is not much different than developing applications for use with other database products. This chapter will briefly introduce the process of authoring client applications for use with ALTIBASE HDB. For more detailed information about writing client applications, please refer to the *Altibase CLI User' Manual*, the *ODBC User' Manual*, the *Precompiler User's Manual*, and the *API User's Manual*.

9.2 Applications Using Altibase CLI

This section explains how to write client applications using Altibase CLI. Altibase CLI is an API that can be used in an environment where ALTIBASE HDB is operated in the client-server structure. For further information, please refer to the *Altibase CLI User's Manual*.

9.2.1 Header Files and Libraries

To develop a program using Altibase CLI, the following files, which can be found in the "include" and "lib" subdirectories of the ALTIBASE HDB home directory, are needed:

- $ALTIBASE_HOME/include/sqlcli.h
- $ALTIBASE_HOME/lib/libodbccli.a

9.2.2 Makefile

In order to compile the Altibase CLI source code in your program, the following must be included in the Makefile:

```
include $(ALTIBASE_HOME)/install/altibase_env.mk
```

This file includes links to library paths and libraries that are needed at compile time, as well as instructions for making object files. Please refer to the sample Makefile in $ALTIBASE_HOME/sample/SQLCLI.

- Makefile Sample Code

```
include $(ALTIBASE_HOME)/install/altibase_env.mk
SRCS=
OBJS=$(SRCS:.cpp=.$(OBJEXT))
BINS=demo_ex1
all: $(BINS)
demo_ex1: demo_ex1.$(OBJEXT)
$(LD) $(LFLAGS) $(LDOUT)demo_ex1$(BINEXT) demo_ex1.$(OBJEXT)
$(LIBOPT)odbccli$(LIBAFT) $(LIBOPT)alticore$(LIBAFT) $(LIBS)
```

9.2.3 Multi-threaded Programming

When developing a multi-threaded program, please keep the following in mind.

- Each thread must have an environment handle, a connection handle, etc. separately allocated thereto.

9.2.4 Writing Applications

The following code shows how to connect to and disconnect from an ALTIBASE HDB server in a program that uses Altibase CLI:

9.2.4.1 Altibase CLI Code Sample

```
/* test.cpp */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include <sqlcli.h>

void sbigint_bigint(int cnt);
void slong_integer(int cnt);
void char_char(int cnt);
void char_number(int cnt);
void double_double(int cnt);
void prepare();
void execute();
void usage();
long logMsec(const char *astr);
void conn(char *port, char *conntype);

#define MSG_LEN 1024
SQLHENV env; // Handle for the environment.
SQLHDBC con; // Handle for the connection.
SQLHSTMT hstmt; // Handle for a statement.
SQLHSTMT bstmt; // Handle for a statement.
int errNo;
short msgLength;
char errMsg[MSG_LEN];
SQLRETURN rc;

/* Main program */
int main(int ac, char **av)
{
 if (ac < 5)
 {
 usage();
 }
 conn(av[2], av[3]);
 switch(atoi(av[1]))
 {
 case 1:
 logMsec(" BIGINT - START TIME : ");
 sbigint_bigint(atoi(av[4]));
 logMsec(" BIGINT - END TIME : ");
 break;
 case 2:
 logMsec(" INTEGER - START TIME : ");
 slong_integer(atoi(av[4]));
 logMsec(" INTEGER - END TIME : ");
 break;
 case 3:
 logMsec(" CHAR - START TIME : ");
 char_char(atoi(av[4]));
 logMsec(" CHAR - END TIME : ");
 break;
 case 4:
 logMsec(" NUMBER - START TIME : ");
 char_number(atoi(av[4]));
 logMsec(" NUMBER - END TIME : ");
 break;
 case 5:
 logMsec(" DOUBLE - START TIME : ");
```

```
 double_double(atoi(av[4]));
 logMsec(" DOUBLE - END TIME : ");
 break;
 }
}

/* print the usage of the program */
void usage()
{
 printf("Usage: ./test <program_no> <port_no> <conntype> <cnt>\n");
 printf("\tprogram_no : 1 => \t SBIGINT-BIGINT\n");
 printf("\tprogram_no : 2 => \t SLONG-INTEGER\n");
 printf("\tprogram_no : 3 => \t CHAR-CHAR\n");
 printf("\tprogram_no : 4 => \t CHAR-NUMERIC\n");
 printf("\tprogram_no : 5 => \t DOUBLE-DOUBLE\n");
 exit(1);
}

/* Check the starting time and the ending time of the program */
long logMsec(const char *astr)
{
 struct timeval tv;
 struct tm *ctm;
 gettimeofday(&tv,NULL);
 ctm = localtime(&(tv.tv_sec));
 fprintf(stderr, "%s [%.02d:%.02d:%.02d]\n", astr, ctm->tm_hour, ctm->tm_min,
ctm->tm_sec);
 return tv.tv_usec;
}

/* Altibase connection statement */
void conn(char *port, char *conntype)
{
 char connStr[200];
 char query[200];
 if (SQL_ERROR == SQLAllocEnv(&env))
 {
 fprintf(stderr, "SQLAllocEnv error!!\n"); //Memory allocation for the envi-
ronment.
 return;
 }
 if (SQL_ERROR == SQLAllocConnect(env, &con)) // Memory allocation for a con-
nection
 {
 fprintf(stderr, "SQLAllocConnect error!!\n");
 SQLFreeEnv(env);
 return;
 }
 sprintf((char*)connStr, "DSN=127.0.0.1;PORT_NO=%s;UID=SYS;PWD=MANager;CONN-
TYPE=%s", port, conntype);

/* Connection creation */
 if (SQL_ERROR == SQLDriverConnect(con, NULL, (SQLCHAR*)connStr, SQL_NTS,
NULL, 0, NULL, SQL_DRIVER_NOPROMPT))
 {
 if (SQL_SUCCESS == SQLError(env, con, NULL, NULL, &errNo, (SQLCHAR*)errMsg,
MSG_LEN, &msgLength))
 {
 fprintf(stderr, " rCM_-%d : %s\n", errNo, errMsg);
 }
 SQLFreeConnect(con);
 SQLFreeEnv(env);
 return;
 }
```

```
/* Not automatically reflected upon execution of each SQL statement. */
 SQLSetConnectAttr(con, SQL_ATTR_AUTOCOMMIT, (void*)SQL_AUTOCOMMIT_OFF, 0);
 if (rc == SQL_ERROR)
 {
 if (SQL_SUCCESS == SQLError(env, con, NULL, NULL, &errNo, (SQLCHAR*)errMsg,
MSG_LEN, &msgLength))
 {
 fprintf(stderr, "[%d : %s]\n", errNo, errMsg);
 }
 }
 hstmt = bstmt = SQL_NULL_HSTMT;
 SQLAllocStmt(con, &hstmt);
 SQLAllocStmt(con, &bstmt);

/* Executing the DDL statement directly and output the message in the defined
format into a file. */
 strcpy(query, "drop table t1");
 rc = SQLExecDirect(hstmt, (SQLCHAR*)query, SQL_NTS);
 if (rc == SQL_ERROR)
 {
 if (SQL_SUCCESS == SQLError(env, con, hstmt, NULL, &errNo, (SQLCHAR*)errMsg,
MSG_LEN, &msgLength))
 {
 fprintf(stderr, "[%d : %s]\n", errNo, errMsg);
}
 }

strcpy(query, "create table t1(i1 number(6))");
 rc = SQLExecDirect(hstmt, (SQLCHAR*)query, SQL_NTS);
 if (rc == SQL_ERROR)
 {
 if (SQL_SUCCESS == SQLError(env, con, hstmt, NULL, &errNo, (SQLCHAR*)errMsg,
MSG_LEN, &msgLength))
 {
 fprintf(stderr, "[%d : %s]\n", errNo, errMsg);
}
 }
}

/* Preparing a SQL statement */
void prepare()
{
 char query[100];
 strcpy(query, "insert into t1 values(?)");
 rc = SQLPrepare(bstmt, (SQLCHAR*)query, SQL_NTS);
 if (rc == SQL_ERROR)
 {
 if (SQL_SUCCESS == SQLError(env, con, bstmt, NULL, &errNo, (SQLCHAR*)errMsg,
MSG_LEN, &msgLength))
 {
 fprintf(stderr, "[%d : %s]\n", errNo, errMsg);
 }
 }
}

/* Executing a prepared SQL statement */
void execute()
{
 rc = SQLExecute(bstmt);
 if (rc == SQL_ERROR)
 {
 if (SQL_SUCCESS == SQLError(env, con, bstmt, NULL, &errNo, (SQLCHAR*)errMsg,
MSG_LEN, &msgLength))
 {
 fprintf(stderr, "[%d : %s]\n", errNo, errMsg);
```

```
 }
 }
}
void sbigint_bigint(int cnt)
{
 int i;
 long long i1;
 char tmp[100];
 int len = SQL_NTS;
 prepare();

/* Binding parameters. */
 SQLBindParameter(bstmt, 1, SQL_PARAM_INPUT, SQL_C_SBIGINT, SQL_BIGINT, 0, 0,
(void*)&i1, 0, &len);
 for(i=0; i<cnt; i++)
 {
 sprintf(tmp, "%d", i);
 i1 = atol(tmp);
 execute();
 }

/* Process COMMIT transaction. */
 rc = SQLTransact(NULL, con, SQL_COMMIT);
}
void slong_integer(int cnt)
{
 int i;
 int i1;
 char tmp[100];
 int len = SQL_NTS;
 prepare();
 SQLBindParameter(bstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0,
(void*)&i1, 0, &len);
 for(i=0; i<cnt; i++)
 {
 sprintf(tmp, "%d", i);
 i1 = atoi(tmp);
 execute();
 }

/* Process COMMIT transaction. */
 SQLTransact(NULL, con, SQL_COMMIT);
}
void char_char(int cnt)
{
 int i;
 char i1[100];
 char tmp[100];
 int len = SQL_NTS;
 prepare();
 SQLBindParameter(bstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
sizeof(i1)-1, 0, (void*)i1, sizeof(i1), &len);
 for(i=0; i<cnt; i++)
 {
 sprintf(tmp, "%d", i);
 strcpy(i1, tmp);
 execute();
 }

/* COMMIT a transaction. */
 SQLTransact(NULL, con, SQL_COMMIT);
}
void char_number(int cnt)
{
 int i;
```

```
 char i1[100];
 char tmp[100];
 int len = SQL_NTS;
 prepare();
 SQLBindParameter(bstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_NUMERIC,
sizeof(i1)-1, 0, (void*)i1, sizeof(i1), &len);
 for(i=0; i<cnt; i++)
 {
 sprintf(tmp, "%d", i);
 strcpy(i1, tmp);
 execute();
 }

/* COMMIT a transaction. */
 SQLTransact(NULL, con, SQL_COMMIT);
}
void double_double(int cnt)
{
 int i;
 double i1;
 char tmp[100];
 int len = SQL_NTS;
 prepare();
 SQLBindParameter(bstmt, 1, SQL_PARAM_INPUT, SQL_C_DOUBLE, SQL_DOUBLE, 0, 0,
(void*)&i1, 0, &len);
 for(i=0; i<cnt; i++)
 {
 sprintf(tmp, "%d", i);
 i1 = atof(tmp);
 execute();
 }

/* COMMIT a transaction. */
 SQLTransact(NULL, con, SQL_COMMIT);
}
```

9.2.4.2 Execution Results

```
$ make test
$ test 1 20300 1 100
BIGINT - START TIME : [16:43:48]
BIGINT - END TIME : [16:43:49]
```

9.3 Applications Using JDBC

The following describes how to create a client application that uses the JDBC driver of ALTIBASE HDB. For more information about the JDBC driver of ALTIBASE HDB, please refer to the *ALTIBASE HDB JDBC User's Manual*.

9.3.1 JDBC Driver

By default, ALTIBASE HDB provides the JDBC driver file, altibase.jar, in the $ALTIBASE_HOME/lib directory. To connect to an ALTIBASE HDB server, the driver is first loaded, and then an attempt is made to access the URL. The form of URL that is supported by the JDBC driver of ALTIBASE HDB is as follows:

```
jdbc:Altibase://hostname:portnum/databasename
```

Step 1: In order to load the JDBC driver, it must be registered in the program using code like the following:

```
Class.forName("Altibase.jdbc.driver.AltibaseDriver")
```

Step 2: Usually, the URL is provided and an attempt to connect to the URL is made as follows. (In this example, the ID used to log in to ALTIBASE HDB is "SYS", and the password is "manager".)

```
String url = "jdbc:Altibase://127.0.0.1:20300/mydb";
Connection con = DriverManager.getConnection(url, "SYS", "manager");
```

9.3.2 CLASSPATH

To run a java application of ALTIBASE HDB, the Altibase.jar file must be included in CLASSPATH. As an example, to use Altibase.jar, CLASSPATH is set as follows in the default login shell files (e.g. .bashrc, .profile, etc.), which can be found in the user's home directory. (This example uses a Bourne shell environment and assumes that java 1.2, which provides a compiler, tools, runtime environment, APIs, etc. has been installed under the /usr/ directory in a UNIX environment.)

```
export JAVA_HOME=/usr/java1.2
export CLASSPATH=$ALTIBASE_HOME/lib/Altibase.jar:$CLASSPATH
```

9.3.3 Writing Applications

This simple program code shows you how to connect to and disconnect from a database using JDBC APIs of ALTIBASE HDB.

9.3.3.1 JDBC Code Sample

```
/* JdbcTest.java */
import java.util.Properties;
import java.sql.*;
class JdbcTest
{
 public static void main(String args[]) {
```

```
Properties props = new Properties();
Connection con = null;
Statement stmt = null;
PreparedStatement pstmt = null;
ResultSet res;

if ( args.length == 0 )
{
System.err.println("Usage : java JdbcTest port_no\n");
System.exit(1);
}

String port = args[0];
String url = "jdbc:Altibase://127.0.0.1:" + port + "/mydb";
String user = "SYS";
String passwd = "MANager";
String enc = "US7ASCII";

props.put("user", user);
props.put("password", passwd);
props.put("encoding", enc);

/* Register Altibase JDBC driver*/
try {
Class.forName("Altibase.jdbc.driver.AltibaseDriver" );
} catch ( Exception e ) {
System.err.println("Can't register Altibase Driver");
return;
}

/* Allocate statement after connection. */
try {
con = DriverManager.getConnection(url,props);
stmt = con.createStatement();
} catch ( Exception e ) {
e.printStackTrace();
}

/* Query */
try {
stmt.execute("DROP TABLE TEST001");
} catch ( SQLException se ) { }

try {
stmt.execute("CREATE TABLE TEST001 ( name varchar(20), age number(3) )");

pstmt = con.prepareStatement("INSERT INTO TEST001 VALUES(?,?)");

pstmt.setString(1,"Hong Gil-dong");
pstmt.setInt(2,25);
pstmt.execute();

res = stmt.executeQuery("SELECT * FROM TEST001");

/* Output the received results on screen */
while(res.next()) {
System.out.println(" Name : "+res.getString(1)+", Age : "+res.getInt(2));
}

/* Disconnected */
stmt.close();
pstmt.close();
con.close();
} catch ( Exception e ) {
e.printStackTrace();
```

```
  }
  }
}
```

9.3.3.2 Execution Results

```
$ javac JdbcTest
$ java JdbcTest 20300 <- port
Name : Hong Gil-dong, Age : 25
```

9.4 Applications Using ODBC with MS Windows

This section describes how to use the Altibase ODBC driver to develop applications. For further information, please refer to the *ODBC User's Manual*.

9.4.1 Installing and Configuring the ODBC Driver

Please refer to the *Windows ODBC Driver Installer User's Guide* and *ODBC User's Manual* for information on how to install the ALTIBASE HDB ODBC driver and configure a DSN in a Windows environment.

9.5 Applications Written Using the C/C++ Precompiler

ALTIBASE HDB C/C++ precompiler converts source code that contains embedded SQL statements to run-time library calls, and creates a new source program that can be compiled in the host language.

This chapter describes how to develop applications using the ALTIBASE HDB C/C++ precompiler. For more information about the ALTIBASE HDB C/C++ precompiler, please refer to the *ALTIBASE HDB Precompiler User's Manual*.

Environment Settings

The following environment settings must be made in order to compile and link files precompiled using the C/C++ precompiler:

9.5.0.1 Header File

The necessary header file is ses.h, and is located in $ALTIBASE_HOME/include/.

To compile precompiled programs, the following compiler option must be used:

```
-I $ALTIBASE_HOME/include
```

9.5.0.2 Library

The necessary library files are libapre.a and libodbccli.a, which are located in the $ALTIBASE_HOME/lib directory.

To link the precompiled application program, the following options must be used:

```
-L $ALTIBASE_HOME/lib -lapre -lodbccli -lpthread
```

9.5.1 Precompiling

The C/C++ precompiler converts code that was written in C or C++ and contains embedded SQL statements to a C or C++ application.

The input file, which contains the code written in C or C++, has the .sc filename extension, and the output file has the .c or .cpp filename extension. While the default filename extension of the output file is .c, the user can set this freely as desired.

9.5.1.1 Precompiling embedded C/C++ programs

The following example shows the use of various options when precompiling:

```
$ apre -h
=====================================================================
APRE (Altibase Precompiler) C/C++ Precompiler HELP Screen
=====================================================================
Usage  :  apre [<options>] <filename>

-h               : Display this help information.
-t <c|cpp>       : Specify the file extension for the output file.
                   c   - File extension is '.c' (default)
```

```
                    cpp - File extension is '.cpp'
-o <output_path> : Specify the directory path for the output file.
                    (default : current directory)
-mt              : When precompiling a multithreaded application,
                    this option must be specified.
-I<include_path> : Specify the directory paths for files included using APRE
C/C++.
                    (default : current directory)
-parse <none|partial|full>
                 : Control which non-SQL code is parsed.
-D<define_name>  : Use to define a preprocessor symbol.
-v               : Output the version of APRE.
-n               : Specify when CHAR variables are not null-padded.
-unsafe_null     : Specify to suppress errors when NULL values are fetched
                    and indicator variables are not used.
-align           : Specify when using alignment in AIX.
-spill <values>  : Specify the register allocation spill area size.
-keyword         : Display all reserved keywords.
-debug <macro|symbol>
                 : Use for debugging.
                    macro   - Display macro table.
                    symbol  - Display symbol table.
-nchar_var <variable_name_list>
                 : Process the specified variables using
                    the Altibase national character set.
-nchar_utf16     : Set client nchar encoding to UTF-16.

=========================================================================

<filename> : The name of a source file containing embedded SQL statements.
Its filename extension must be .sc.
```

9.5.2 Multi-threaded Programming

The ALTIBASE HDB C/C++ precompiler supports multi-threaded programming. The following are some considerations to keep in mind when developing multi-threaded applications that contain embedded SQL statements:

- The user must indicate to the precompiler that the program is a multithreaded program.
- Each thread must have its own connection.
- The name of each connection must be unique in the program.
- Each connection name within a program must be unique.
- Embedded SQL statements must also indicate the name of the connection they will use.

9.5.3 Writing Applications

9.5.3.1 Apre C/C++ Code Sample

```
/*********************************************************
 * SAMPLE : DELETE
 * .CODE : delete.sc
 * 1. Using scalar host variables
 * 2. Reference : array host variables - arrays1.sc
 *********************************************************/
```

```
int main()
{
 /* declare host variables */
 EXEC SQL BEGIN DECLARE SECTION;
 char usr[10];
 char pwd[10];
 char conn_opt[1024];
 /* scalar type */
 int s_eno;
 short s_dno;
 EXEC SQL END DECLARE SECTION;

 printf("<DELETE>\n");

 /* name, password, options */
 strcpy(usr, "SYS");
 strcpy(pwd, "MANAGER");
 strcpy(conn_opt, "DSN=127.0.0.1;CONNTYPE=1;PORT_NO=20300");

 /* Altibase server connection */
 EXEC SQL CONNECT :usr IDENTIFIED BY :pwd USING :conn_opt;

 /* check sqlca.sqlcode */
 if (sqlca.sqlcode != SQL_SUCCESS)
 {
 printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
 exit(1);
 }

 /* use scalar host variables */
 s_eno = 5;
 s_dno = 1000;
 EXEC SQL DELETE FROM EMPLOYEES
 WHERE ENO > :s_eno
AND DNO > :s_dno
AND EMP_JOB LIKE 'P%';
 printf("-----------------------------------------------\n");
 printf("[Scalar Host Variables] \n");
 printf("-----------------------------------------------\n");

 /* check sqlca.sqlcode */
 if (sqlca.sqlcode == SQL_SUCCESS)
 {
 /* sqlca.sqlerrd[2] holds the rows-processed(deleted) count */
 printf("%d rows deleted\n\n", sqlca.sqlerrd[2]);
 }
 else
 {
 printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
 }

 /* disconnect */
 EXEC SQL DISCONNECT;

 /* check sqlca.sqlcode */
 if(sqlca.sqlcode != SQL_SUCCESS)
 {
 printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
 }
}
```

9.5.3.2 Execution Result

```
$ make delete
$ delete
<DELETE>
-----------------------------------------------------
[Scalar Host Variables]
-----------------------------------------------------
7 rows deleted
```

Index