Altibase Administration

Replication Manual

Release 6.1.1

May 16, 2012

ALTIBASE
PERFORMANCE SOLUTIONS

Altibase Administration Replication Manual
Release 6.1.1

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

Contents

Preface

About This Manual

This manual gives an overview of replication in ALTIBASE® HDB™ and explains in detail how to perform replication.

Target Users

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

- database administrators
- application designers
- programmers

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 using relational databases and an understanding of database concepts
- computer programming experience

Software Environment

This manual has been prepared assuming that ALTIBASE HDB 5.5.1 will be used as the database server.

Organization

This manual has been organized as follows:

- Chapter1: Replication Overview

  This chapter introduces replication in ALTIBASE HDB.

- Chapter2: Managing Replication

  This chapter explains replication procedures in ALTIBASE HDB.

- Chapter3: Deploying Replication

  This chapter explains how to establish a replication environment in ALTIBASE HDB.

- Chapter4: Fail-Over

  This chapter explains the Fail-Over feature provided with ALTIBASE HDB and how to use it.

- Appendix A. FAQ

Documentation Conventions

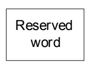
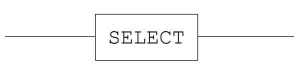
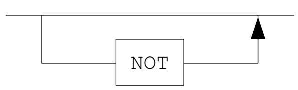
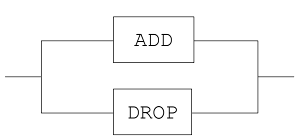
This chapter describes the conventions used in this manual. Understanding these conventions will make it easier to find information in this manual and other manuals in the series.

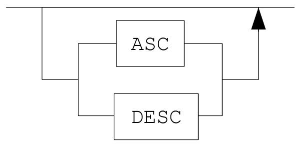
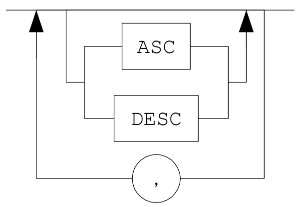
There are two sets of conventions:

- syntax diagrams
- sample code conventions

Syntax Diagrams

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

Elements	semantics
	The start of a command. If a syntactic element starts with an arrow, it is not a complete command.
	The command continues to the next line. If a syntactic element ends with this symbol, it is not a complete command.
	The command continues from the previous line. If a syntactic element starts with this symbol, it is not a complete command.
	The end of a statement.
	Indicates a mandatory element.
	Indicates an optional element.
	Indicates a mandatory element comprised of options. One, and only one, option must be specified.

Elements	semantics
	Indicates an optional element comprised of options.
	Indicates an optional element in which multiple elements may be specified. A comma must precede all but the first option.

Sample Code Conventions

The code examples explain SQL, stored procedures, iSQL, and command-line statements.

The printing conventions used in the code examples are described in the following table.

<table>
<tr><th>Convention</th><th>Meaning</th><th>Example</th></tr>
<tr><td>[ ]</td><td>Indicates an optional item.</td><td><code>VARCHAR [(size)] [[FIXED |] VARIABLE]</code></td></tr>
<tr><td>{ }</td><td>Indicates a mandatory field for which one or more items must be selected.</td><td><code>{ ENABLE | DISABLE | COMPILE }</code></td></tr>
<tr><td>|</td><td>A delimiter between optional or mandatory arguments.</td><td><code>{ ENABLE | DISABLE | COMPILE }</code>
<code>[ENABLE | DISABLE | COMPILE]</code></td></tr>
<tr><td>.
.
.</td><td>Indicates that the previous argument is repeated, or that sample code has been omitted.</td><td><code>iSQL> select e_lastname from employees;</code>
<code>E_LASTNAME</code>
<code>------------------------</code>
<code>Moon</code>
<code>Davenport</code>
<code>Kobain</code>
<code>.</code>
<code>.</code>
<code>.</code>
<code>20 rows selected.</code></td></tr>
<tr><td>Other symbols</td><td>Symbols other than those shown above are part of the actual code.</td><td><code>EXEC :p1 := 1;</code>
<code>acc NUMBER(11,2);</code></td></tr>
</table>

Convention	Meaning	Example
Italics	Statement elements in italics indicate variables and special values specified by the user.	`SELECT * FROM` *`table_name`*`;` `CONNECT` *`userID/password`*`;`
Lower Case Characters	Indicate program elements set by the user, such as table names, column names, file names, etc.	`SELECT e_lastname FROM employees;`
Upper Case Characters	Keywords and all elements provided by the system appear in upper case.	`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 Getting Started
- ALTIBASE HDB SQL Reference
- ALTIBASE HDB iSQL User's Manual
- ALTIBASE HDB Error Message Reference

On-line Manuals

Manuals are available at the Altibase Technical Center (http://atc.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

Write to us at the following e-mail address : support@altibase.com

For immediate assistance regarding technical issues, please contact the Altibase Technical Center.

Thank you. We appreciate your feedback and suggestions.

1 Replication Overview

1.1 Introduction

The purpose of database replication is to maintain an up-to-date backup of the data on an Active Server and provide an uninterrupted service environment in which a substitute server can be used to resume service in the event that the Active Server unexpectedly goes offline for some reason.

This chapter covers the following subjects:

- ALTIBASE HDB Replication Concepts and Terminology
- How to Perform Replication in ALTIBASE HDB
- Choosing Replication Targets
- Replication Mode
- Replication of Partitioned Tables
- Data Recovery Using Replication

1.1.1 Concepts

The basic idea behind replication in ALTIBASE HDB is the use of the log replay method. To support the replication feature of ALTIBASE HDB, a local server transfers transaction logs to a remote server when the logs change. The remote server "replays" the received logs to its database, that is, it implements the changes that have been recorded in the logs. ALTIBASE HDB also provides the Audit utility for monitoring and managing the status of replication.

1.1.2 Terminology

Active Server

This is a replication node that is actively providing service to users, and on which change operations related to master transactions are taking place.

Change Operation

This term indicates an INSERT, UPDATE, or DELETE DML operation. This term is necessary in order to distinguish these operations from SELECT operations, which do not change the contents of a database.

EAGER Mode

This is one of two available replication modes, and prioritizes data consistency over performance. In this mode, a transaction is not committed on the local server until the local server receives a message from the remote server stating that the task has been performed and the transaction replayed on the remote server.

LAZY Mode

This is the other of the two available replication modes, and prioritizes performance over data consistency. In this mode, a transaction is committed on the local server without waiting for confirmation from the remote server.

Local Commit XSN

This is the sequence number of the committed log record that was most recently read by the Sender. The transaction corresponding to this XSN is not guaranteed to have been committed on the remote server. This value is returned when the COMMIT_XSN column of the V$REPSENDER performance view is queried.

Local Server

In this manual, the term "local server" always refers to the local node, that is, to the server on which the current task is being performed, regardless of whether it is an Active or Standby Server, or whether it hosts a replication Sender or Receiver thread.

Master Transaction

This is a transaction that takes place on an active server in the course of providing service to users. It involves the execution of one or more change (INSERT, UPDATE, or DELETE) operations on one or more replication target tables.

Parallel Replication

This is the use of multiple Sender and Receiver threads to perform replication in EAGER mode. This is not to be confused with parallel synchronization.

Parallel Synchronization

This is the use of multiple Sender and Receiver threads to accomplish a synchronization task (using "ALTER REPLICATION ... SYNC" or "ALTER REPLICATION ... SYNC ONLY"). This is not to be confused with parallel replication.

Receiver

This is a thread that receives XLogs, which contain information about changes to data, from a counterpart server, and replays the XLogs on replication target objects on the local node.

Receiver Thread

This has the same meaning as "Receiver" when not using parallel replication, i.e. when performing replication using only one Sender thread and one Receiver thread. When using parallel replication, one Receiver comprises multiple Receiver threads.

Remote Server

This is a counterpart replication node, i.e. a node that has a 1:1 relationship with a local server to form a replication pair.

Replication

This term indicates the concept and action of replicating, rather than a concrete object or entity.

Replication Gap

Conceptually, the replication gap is an indicator of how far the replication process has fallen behind the current state of the database. In quantitative terms, it is the difference between the sequence number (not XSN) of the most recent redo log and the sequence number of the redo log for which the corresponding XLog is currently being sent.

Replication Manager

This is the ALTIBASE HDB module that starts and stops the replication Sender and Receiver.

Replication Object

This is an object created with the CREATE REPLICATION statement. It forms a replication pair with a counterpart replication object on another node.

Replication Pair

This is a pair of corresponding replication objects having the same name, one residing on each of two different nodes.

Replication Target Column

This is a column that exists in corresponding replication target tables on local and remote servers. Replication target columns cannot be explicitly designated; rather, they are determined based on the structure of the corresponding replication target tables.

Replication Target Table

This is a table that is designated, using the CREATE REPLICATION or ALTER REPLICATION statement, to be replicated between corresponding replication nodes.

Replication Transaction

This is a transaction that replicates a master transaction on another server. It replayes the execution of one or more change (INSERT, UPDATE, or DELETE) operations on one or more replication target tables. It occurs when the Receiver receives an XLog.

Restart SN

This is the lowest Redo SN (not XSN) corresponding to a transaction for which an XLog for replication has not been sent. It is the position from which the transmission of XLogs will recommence when replication resumes.

Sender

This is a thread that sends information about changes made to data by a transaction to a remote server. It changes logs that result from the execution of DML statements on replication target tables on the local server into XLog form so that they contain information about the actual (physical) changes made to the data, and sends the resultant XLogs to the remote server.

Sender Thread

This has the same meaning as "Sender" when not using parallel replication, i.e. when performing replication using only one Sender thread and one Receiver thread. When using parallel replication, one Sender comprises multiple Sender threads.

Standby Server

This is a replication node on which change transactions are not occurring. (It may be queried using SELECT DML statements.)

Synchronization

"Synchronization" is a unidirectional operation in which all data in the replication target tables on the local server are inserted ino the corresponding tables on the remote server. If any data conflict occurs on the remote server during synchronization, conflict resolution will be applied on the remote server. It is performed by executing the ALTER REPLICATION DDL statement with either the SYNC or SYNC ONLY keyword.

XLog

This is a kind of log that results from the transformation of one or more redo logs into logical form for replication. The replication Sender thread on a local server transmits an XLog to the replication Receiver thread on a remote server, which replays the log so that the remote server contains the same data as the local server.

XSN

This stands for "XLog Sequence Number". It is not to be confused with "SN", which is the sequence number of a redo log.

1.1.3 How to Perform Replication in ALTIBASE HDB

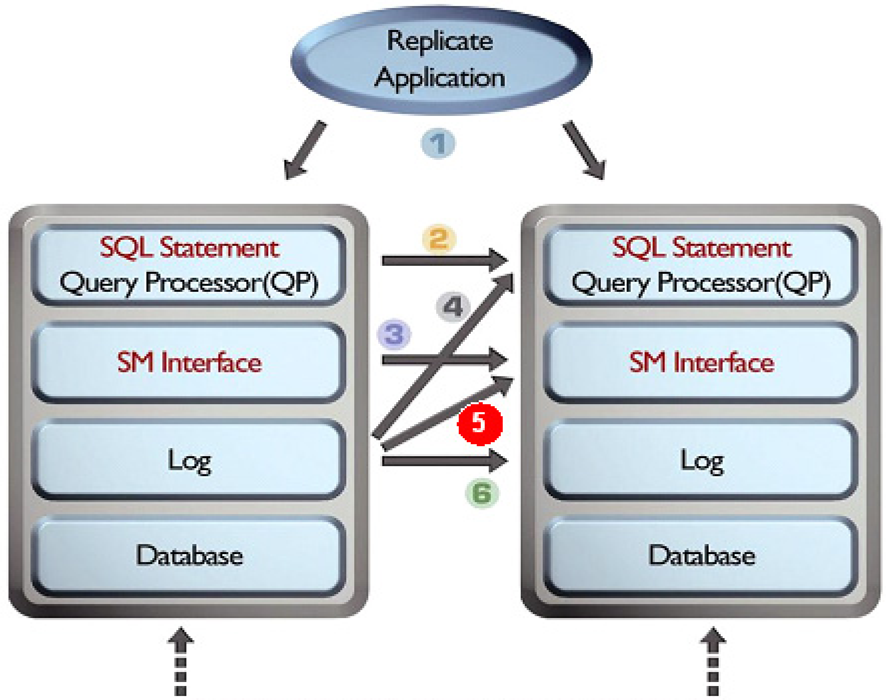
Replication is conducted in this way: the local server sends information about changes to database contents to the remote server, and then the remote server makes the same changes to its database.

Thus, in addition to service threads, the local and remote servers operate additional threads required to manage replication.

The replication Sender thread on the local server sends information about changes to database contents to the remote server, and then the replication Receiver thread on the remote server makes the same changes to the database on the remote server. Additionally, the replication Sender and Receiver threads also automatically detect whether the corresponding servers shut down normally or abnormally, and then perform appropriate tasks.

ChapterFigure 1-1: A Review of the Methods of Replication illustrates various ways that replication is supported. In ALTIBASE HDB, in consideration of performance and flexibility, the best of these ways is to transform redo logs into a directly executable logical structure.

Figure 1-1 A Review of the Methods of Replication



1. Performing Replication using a Client Application

   When using this method, performance can suffer, and it is difficult to ensure data consistency.

   Because replication is log-based in ALTIBASE HDB, using an application to issue commands to perform replication makes it difficult to ensure data consistency because duplicate queries must be run, and because issues arise with respect to the order in which transactions are conducted.

2. Sending Queries

   When using this method, the load on the QP (Query Processor) is increased, and validation is difficult due to the occurrence of data collisions.

3. Sending Execution Plans

   When using this method, the communication load is increased due to the increased volume of transmissions.

4. Converting Logs into Query Statements

   This method incurs high conversion and query processing expense.

5. Converting Logs Directly into a Form that Can Be Executed

   This method incurs high conversion expense but improves replication performance.

6. Transmitting Logs and Performing Log-Based Recovery

This method is fast, but cannot be used in an "Active-Active" environment (one in which both servers are providing service).

1.1.4 Choosing a Replication Server

In order to conduct replication in ALTIBASE HDB, the database character sets and the national character sets must be the same on both the local and remote servers. The character set can be checked by querying the V$NLS_PARAMETERS performance view.

1.1.5 Choosing Replication Targets

ALTIBASE HDB uses object names to specify replication targets. When defining a replication, the names of users and tables that are to be designated as replication targets must be directly specified. Additionally, only columns that have the same names on both the local and remote servers at the time of replication can be replication targets.

The replication target columns can be checked by querying the V$REPRECEIVER_COLUMN performance view.

1.1.6 Replication Mode

In ALTIBASE HDB, replication can run in one of the following modes:

- LAZY Mode
- EAGER Mode

In Table 1-1, each replication mode is characterized according to performance, to whether the possibility of delayed replication exists, and to the level of data consistency.

Table 1-1 Replication Mode

Mode	Performance	Delayed Replication	Data Consistency
LAZY	High	Possible	Low
EAGER	Medium	Impossible	High

1.1.6.1 LAZY Mode

In LAZY mode, when a transaction occurs on a local server ("Master Transaction"), and thus a DML statement is executed on a table that is a replication target, the Sender thread collects logs recorded by the Master Transaction, and then converts them into XLOGs and sends them out. The Receiver thread on the remote server receives these XLOGs and commits the replication transactions to its DB.

Therefore, because the master transaction and the replication transaction take place in complete isolation from one another, the transactions do not influence one another, and the performance of the local server is excellent.

However, since the Sender thread always tracks the master transactions, replication may not always be completely up-to-date on very busy sites.

1.1.6.2 EAGER Mode

In EAGER mode, when a master transaction occurs on a local server, the local server commits the transaction only after it has received confirmation that all of the corresponding logs have been properly applied on the remote server. The remote server commits the replication transaction at the same time. In other words, replication in EAGER mode is a synchronization method[1].

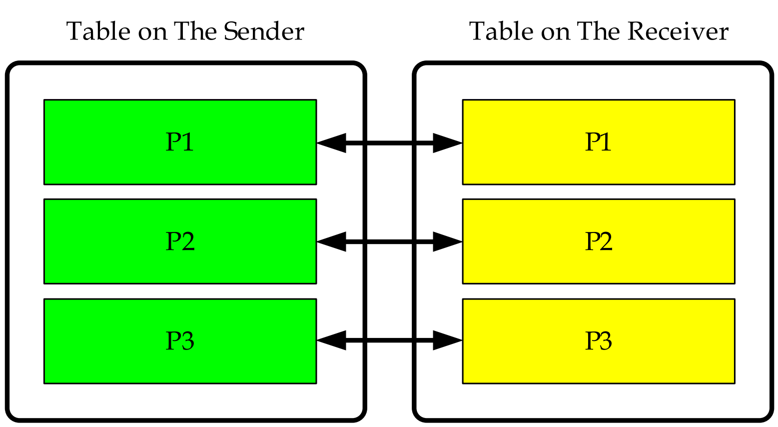
The benefit of EAGER mode is that it is possible to replicate transactions in parallel, because they are synchronized. Therefore, when replication is running in EAGER mode, multiple Sender threads can operate in parallel. The number of parallel threads is set using the REPLICATION_EAGER_PARALLEL_FACTOR property.

In EAGER mode, although performance suffers somewhat due to transaction synchronization, replication is not delayed on very busy servers, which can occur when running in LAZY mode.

1.1.7 Replication of Partitioned Tables

As shown in ChapterFigure 1-2: The Structure of a Replicated Partitioned Table, when a partitioned table is replicated, the replicated table appears from the outside to have the same structure as the original table. Internally, the structure of each partition is also replicated.

Figure 1-2 The Structure of a Replicated Partitioned Table



1. Transaction Synchronization: Even if a master transaction is successfully performed on a local server, if a replication conflict occurs on a remote server, it will be impossible to commit the master transaction on the local server.
In such cases, the user must explicitly roll back the transaction in order to be able to execute the next transaction. If the transaction is not rolled back, it will be impossible to apply any changes because a transaction that cannot be committed is continually pending. Under conditions in which the local server is internally required to commit a transaction, such as when running in Autocommit mode or when a session is terminated, the conflict causes the master transaction that could not be committed to be automatically rolled back.
As a result, the master transaction that experienced the conflict and the replication transaction are both rolled back, which prevents data inconsistencies attributable to replication.

1.1.8 Data Recovery Using Replication

ALTIBASE HDB supports a data recovery option that uses replication to prevent data on replicated servers from becoming mismatched. If a server shuts down abnormally while replication is active, the user can take advantage of this method to recover data using the logs of master transactions that were executed on a normally operating server, or even using the logs of replication transactions.

When the data durability level is set to a lower level in the interests of high performance, data can be synchronized using the replication recovery option to ensure that no committed transactions disappear if a system shuts down abnormally.

1.1 Introduction

2 Managing Replication

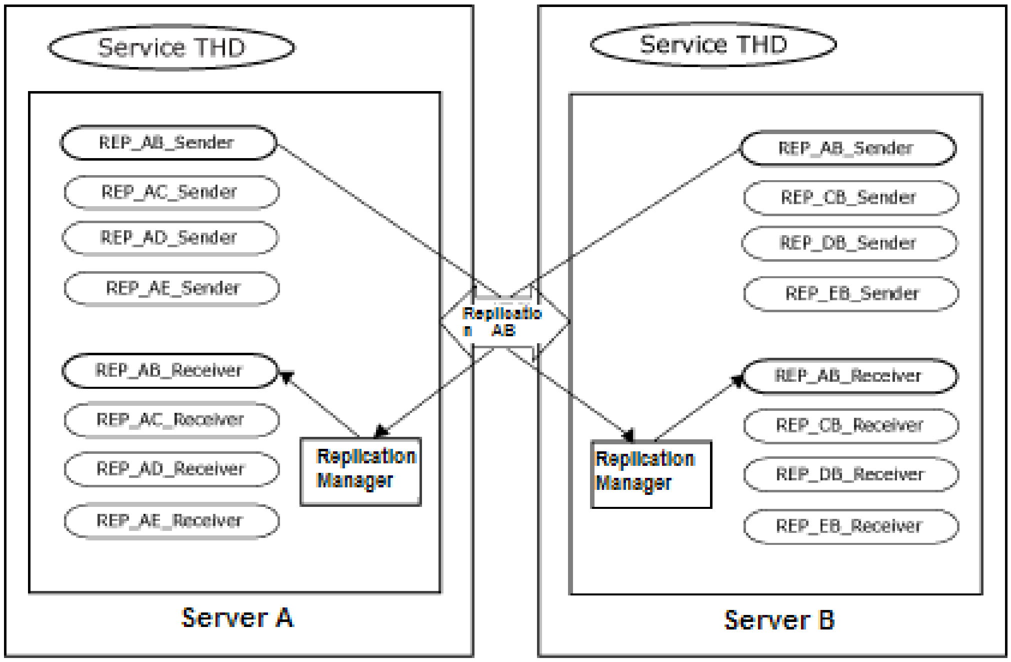
This chapter sets forth the replication steps in order, and explains how to operate replication functions of ALTIBASE HDB in response to various kinds of faults and errors that can occur while replication is active.

This chapter contains the following sections:

2.1 Replication Procedures

The following figure shows how replication works in ALTIBASE HDB.

Figure 2-1 Replication Procedures



1. Choose replication target servers.

   The database character sets and the national character sets must be the same on both servers.

2. Choose tables to be replicated.

   Every table to be replicated must have a primary key.

3. Set replication conditions.

   Set only the logs that pertain to the replication conditions as replication targets. If no replication conditions are specified, all of the data in a table will be the replication target.

4. Create a replication object using the CREATE REPLICATION statement.

   The replication object must have the same name in both databases.

5. Start replication using the ALTER REPLICATION statement.

   When replication is started, the local server creates a replication Sender thread, and this thread connects to a replication manager on the remote server. At this time, the replication manager on the remote server generates a replication Receiver thread.

6. The replication service is started.

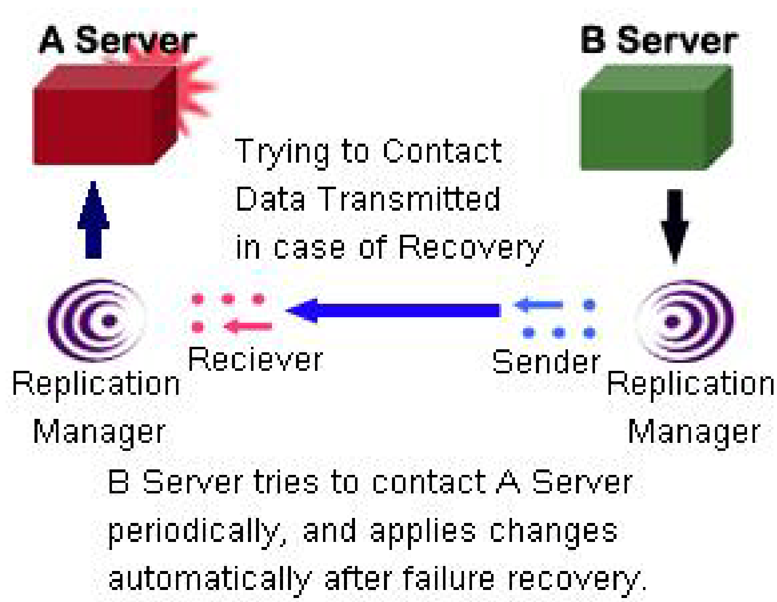
2.2 Troubleshooting

The problems that typically affect replication are:

- Abnormal shutdown of the local or remote server
- Interruption of communication between the local and remote servers
- Network failure

2.2.1 Abnormal shutdown of the local server

Figure 2-2 Replication in the event of server failure



- Abnormal termination of Server A

  The Receiver thread on Server B terminates, and the Sender thread on Server B attempts to connect to Server A at regular intervals (e.g. every 60 seconds).

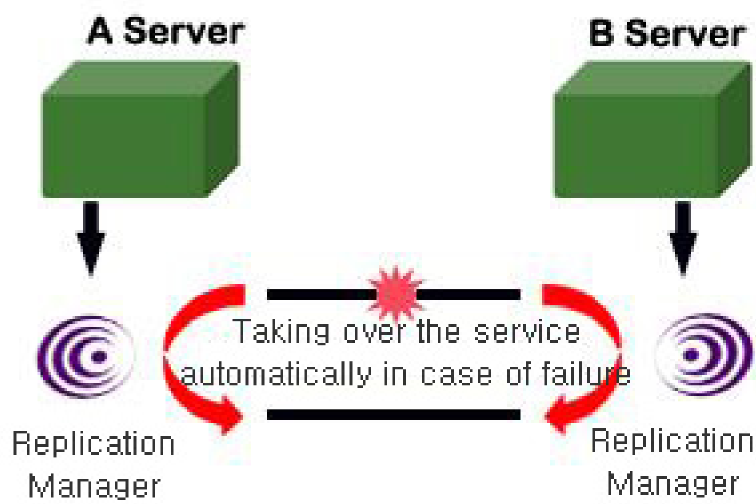
- Restart of Server A (the Sender thread calls the Receiver thread on the remote server)

  1. The Sender thread on Server A automatically starts and performs replication with Server B.
  2. The replication Receiver thread on Server A is started by the Sender thread on Server B, and performs replication.
  3. The Sender thread on Server B starts the Receiver thread on Server A.
  4. The Receiver thread on Server B starts replication after being started by the Sender thread on Server A.

2.2.2 Interruption of communication between the local and remote

servers

Figure 2-3 Replication in response to communication failure with remote server

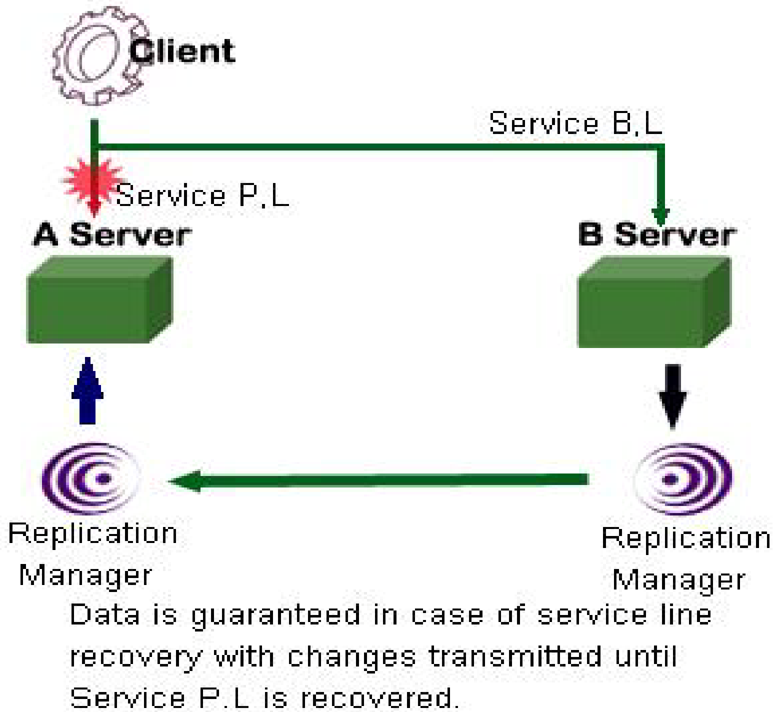


- Communication failure between the local and remote servers
  1. The Receiver threads on Server A and B roll back and terminate uncommitted transactions.
  2. The Sender threads on Server A and B record the [1]Restart SN and attempt to connect to the corresponding servers at intervals of 60 seconds.
- Connection Restoration
  1. The Sender threads on Server A and B wake up the receiver threads on the corresponding servers and perform replication by transmitting all XLOGs, starting with the XLOG corresponding to the Redo Log having the Restart SN.
  2. Receiver threads on Server A and B are created in response to connection requests from the Sender threads on corresponding servers, and perform replication.

1. The term "Restart SN" is defined in the Glossary in Chapter 1.

2.2.3 Network Failure

Figure 2-4 Replication in the event of network failure



- Primary Line Disconnection

  1. Service is provided from Server B using a backup line.

- Primary Line Restoration

  1. Service is provided from Server A again after the primary line is restored.
  2. Even while the primary line is down, Server B can send task contents to Server A using the replication feature of ALTIBASE HDB.

2.3 Confilict Resolution

The term "Data Conflict" refers to the case where records having the same primary key or records to which constraints apply in replication target tables in two or more database servers are changed by respective local transactions.

The best way of completely avoiding data conflicts when using Deferred Replication is to distribute the updated Data Set between systems.

What follows is the three kinds of data conflict and the reasons why each kind of conflict occurs.

- INSERT Conflict
  - When a replication transaction attempts to insert data having the same primary key as an existing record, an insert conflict occurs.
  - If a table into which records are to be inserted for replication has already been locked by a local transaction, a replication transaction must wait to acquire a lock, and an insert conflict can occur due to a lock timeout.
  - When a replication transaction attempts to insert a value that already exists into a column for which a unique constraint has been defined, an insert conflict occurs.
- UPDATE Conflict
  - When an attempt is made to update a record having a nonexistent primary key, an update conflict occurs.
  - When an attempt is made to update a record having a value different from the so-called "Before Image" (i.e. the data as they existed before being updated) on another database server, from which data for replication are propagated, an update conflict occurs.
  - If a record that is to be updated via replication has already been locked by a local transaction, the replication transaction must wait to acquire a lock and an update conflict can occur due to a lock timeout.
  - If an existing record is updated such that a duplicate unique key is generated, an update conflict occurs.
- DELETE Conflict
  - When a replication transaction attempts to delete a record having a nonexistent primary key, a delete conflict occurs.
  - If a record that is to be deleted in the course of replication has already been locked by a local transaction, a replication transaction must wait to acquire a lock, and a delete conflict can occur due to a lock timeout.

Usually, unlike distributed DBMS, which use 2-Phase Commit (2-PC) or 3-Phase Commit (3-PC), in typical DBMSs, there is no way of guaranteeing that data inconsistencies will not be caused by replication-related conflicts. Distributed DBMSs guarantee the consistency of data, but 2-PC and 3-PC entail decreases in performance, and moreover, additional steps must be taken in the event of system or network failure.

Therefore, in order to overcome the limitations related to data consistency with typical DBMSs and maintain their performance at the same time, Deferred (Asynchronous) Replication is commonly used.

The term "conflict resolution" can refer to a variety of methods for eliminating data conflict. Deferred

Replication does not offer a perfect solution to data conflicts. After a conflict has occurred, the conflict is resolved with the goal of harmonizing the data between the database servers. ALTIBASE HDB provides the following conflict resolution methods to resolve data conflicts:

- User-Oriented Scheme
- Master-Slave Scheme
- Timestamp-Based Scheme

In all of the above methods, ALTIBASE HDB performs the following actions:

- changing the data on one server so that they are the same as the data on the other server
- logging information about the conflict so that problems can be tracked

However, data conflicts affecting LOB columns cannot be resolved. This is because "Before Image" logging is not performed for LOB columns, and because primary and unique keys cannot be designated for LOB columns, making it impossible to detect conflicts.

The policies governing every set of conditions under which data conflicts can occur will be discussed in detail below.

Note: For a detailed description of the CREATE REPLICATION command, please refer to the description of the CREATE REPLICATION statement.

2.3.1 User-Oriented Scheme

2.3.1.1 Syntax

```
CREATE REPLICATION replication_name
  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;
```

2.3.1.2 Description

1. INSERT Conflict

   If an insert conflict occurs, the INSERT statement fails, and a conflict error message is recorded in altibase_rp.log.

   The REPLICATION_INSERT_REPLACE property is used to set the policy for resolving the type of conflict that occurs when an attempt is made to insert data having the same primary key as an existing record.

   REPLICATION_INSERT_REPLACE=1 : Delete and insert

   REPLICATION_INSERT_REPLACE=0 : Do not delete or insert; rather, output a conflict error message

2. UPDATE conflict

   When an update conflict occurs, the UPDATE statement fails, and a conflict error message is recorded in altibase_rp.log.

   The REPLICATION_UPDATE_REPLACE property is used to resolve the type of conflict that occurs when an attempt is made to update a record having a value not equal to the “Before Image” on another database server, from which data were propagated.

   For example, suppose that a particular data item is equal to 10, and that a transaction attempts to update that value from 20 to 30. Either of the following policies can be implemented according to the application.

   REPLICATION_UPDATE_REPLACE=1 : Update

   REPLICATION_UPDATE_REPLACE=0 : Do not update, and output a conflict error message

3. DELETE Conflict

   If a delete conflict occurs, the DELETE statement fails, and a conflict error message is recorded in altibase_rp.log.

2.3.1.3 Summary

1. The user can decide whether to commit UPDATEs on a case-by-case basis.

2. The Audit utility provides another solution for dealing with data inconsistencies. For more detailed information, please refer to the Audit User’s Manual.

2.3.2 Master-Slave Scheme

2.3.2.1 Syntax

```
CREATE REPLICATION replication_name AS {MASTER|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;
```

2.3.2.2 Description

1. Specify ”MASTER” or “SLAVE” in the command to specify whether the server is the Master or the Slave. If not specified, the value specified using the REPLICATION_INSERT_REPLACE or REPLICATION_UPDATE_REPLACE property will be used.

2. You can check whether a server is the Master or the Slave by checking the CONFLICT_RESOLUTION field, which is located in the SYS_REPLICATIONS_ meta table. (0 = not specified; 1 = Master; 2 = Slave)

3. When attempting to perform handshaking[1], the following combinations of CONFLICT_RESOLUTION field values will be successful: 0 with 0, 1 with 2, and 2 with 1. Other

combinations will fail. In other words, if one server is set as the Master and the value is not specified on the other server, the following error will be output when replication starts:

```
iSQL> ALTER REPLICATION rep1 START;
[ERR-6100D : [Sender] Failed to handshake with the peer server (Master/
Slave conflict resolution does not allowed [1:0])]
```

2.3.2.3 Master/Slave Replication Conflict Handling Method

1. Operating as Master

- INSERT conflict: Not committed.
- UPDATE conflict: Not committed.
- DELETE conflict: Not committed.
- Other: XLOG transferred from the Slave is processed as usual.

2. Operating as Slave

- INSERT conflict: If an insert conflict occurs because an attempt was made to insert data having the same primary key as an existing record, the existing record is deleted and a new record is added.

  If an insert conflict occurs for any other reason, the INSERT statement fails, and a conflict error message is recorded in altibase_rp.log.
- UPDATE conflict: If an update conflict occurs because an attempt was made to update a record having a value different from the "Before Image" value on another database server, from which data for replication are propagated, the conflict is ignored, and the UPDATE statement succeeds despite the conflict.

  If an update conflict occurs for any other reason, the UPDATE statement fails, and a conflict error message is recorded in altibase_rp.log.
- DELETE conflict: If a delete conflict occurs because no record having that primary key exists, the DELETE statement fails, and a conflict error message is not recorded in altibase_rp.log.

  If a delete conflict occurs for any other reason, the DELETE statement fails, and a conflict error message is recorded in altibase_rp.log.
- Other: The XLOG transferred from the Master is processed as usual.

2.3.2.4 Example

Suppose that the IP address and replication port number of the local server are 192.168.1.10 and 21300, and that the IP address and replication port number of the remote server are 192.168.1.20 and 22300, that there is a master-slave relationship between the local and remote servers, and that a table called *employees* and one called *departments* are replication target tables. In this situation, replication is specified as follows:

- Local Server (IP: 192.168.1.10)

```
iSQL> CREATE REPLICATION rep1 AS MASTER
 WITH '192.168.1.20',22300
```

1. Handshaking is a process for checking whether the other server is alive and whether the information about the objects to be replicated between the local server and the remote server matches before replication starts.

```
 FROM sys.employees TO sys.employees,
 FROM sys.departments TO sys.departments;
Create success.
```

- Remote Server (IP: 192.168.1.20)

```
iSQL> CREATE REPLICATION rep1 AS SLAVE
 WITH '192.168.1.10',21300
 FROM sys.employees TO sys.employees,
 FROM sys.departments TO sys.departments;
Create success.
```

Whether a server is a Master or Slave can be determined by checking the CONFLICT_RESOLUTION field, which is located in the SYS_REPLICATIONS_ meta table. (0 = not specified; 1 = Master; 2 = Slave)

```
iSQL> SELECT replication_name, conflict_resolution FROM
system_.sys_replications_;
REPLICATION_NAME                        CONFLICT_RESOLUTION
-----------------------------------------------------------------
REP1                                    1
1 row selected.
```

2.3.3 Timestamp-Based Scheme

2.3.3.1 Syntax

```
CREATE REPLICATION replication_name
  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;
```

2.3.3.2 Description

The Timestamp-Based Scheme is provided to ensure that both servers have the same data in an Active-Active replication environment.

The following restrictions apply when using the Timestamp-Based Scheme:

- Every table must contain a TIMESTAMP column.
- The REPLICATION_TIMESTAMP_RESOLUTION property must be set to 1.

Because ALTIBASE HDB supports the Timestamp-Based Scheme on the basis of tables, even if a replication target table has a TIMESTAMP column, if the value of the REPLICATION_TIMESTAMP_RESOLUTION property for that table has been set to 0, a conventional conflict resolution scheme will be used.

Supposing for example that a user wishes to replicate a table called "foo" and another called "bar" between two servers, if the REPLICATION_TIMESTAMP_RESOLUTION property is set to 1 for the "foo" table, the Timestamp-Based Scheme will be used for that table, whereas a conventional conflict resolution scheme will be used for the "bar" table.

```
CREATE TABLE foo(a DOUBLE PRIMARY KEY, b TIMESTAMP);
```

```
CREATE TABLE bar(a DOUBLE PRIMARY KEY, b CHAR(3));
CREATE REPLICATION rep WITH '127.0.0.1', 20300 FROM sys.foo TO sys.foo, FROM
sys.bar TO sys.bar;
```

2.3.3.3 Timestamp-based Replication Processing Method

ALTIBASE HDB supports the Timestamp-Based Scheme only for INSERT and UPDATE operations.

- INSERT
  1. If data to be inserted have the same key as existing data, the timestamp value of the After-Image of the data is compared with that of the existing data.
  2. If the TIMESTAMP value of the After-Image of the data is equal to or greater (i.e. more recent) than that of the existing data, the existing data are deleted, and new data, having the value of the After-Image of the data, are added.
- UPDATE
  1. The TIMESTAMP value of the After-Image of the data is compared with that of the data to be updated.
  2. If the TIMESTAMP value of the After-Image of the data is equal to or greater (more recent) than that of the existing data, the data are updated with the After-Image of the data.
  3. When UPDATE is performed, the TIMESTAMP value in the After-Image of the data is kept. In other words, independent system time values are not used.

2.3.3.4 Restrictions

1. When a TIMESTAMP column is added to a table, 8 additional bytes of storage space are needed per record.
2. If the time is set differently on the two servers to be replicated, database inconsistencies can result.

2.4 Related Performance Views

The following performance views are used to monitor the progress of replication:

- V$REPEXEC
- V$REPGAP
- V$REPGAP_PARALLEL
- V$REPLOGBUFFER
- V$REPOFFLINE_STATUS
- V$REPRECEIVER
- V$REPRECEIVER_COLUMN
- V$REPRECEIVER_PARALLEL
- V$REPRECEIVER_STATISTICS
- V$REPRECEIVER_TRANSTBL
- V$REPRECEIVER_TRANSTBL_PARALLEL
- V$REPRECOVERY
- V$REPSENDER
- V$REPSENDER_PARALLEL
- V$REPSENDER_STATISTICS
- V$REPSENDER_TRANSTBL
- V$REPSENDER_TRANSTBL_PARALLEL
- V$REPSYNC

For detailed information on each performance view, please refer to the *ALTIBASE HDB Administrator's Manual*.

3 Deploying Replication

This chapter contains the following sections:

3.1 Considerations

A number of conditions apply when establishing replication. If these conditions are not satisfied, replication will not be possible.

3.1.1 Prerequisites

1. If a conflict occurs during an INSERT, UPDATE, or DELETE operation, the operation is skipped, and a message is written to an error file.
2. If an error occurs during replication, partial rollback is performed. For example, if a duplicate row is found while inserting rows into a table, only the insertion of the duplicate row is cancelled, while the remainder of the task is completed as usual.
3. Replication is much slower than the main data provision service.

3.1.2 Data Requirements

1. A table to be replicated must have a primary key.
2. The primary key must not have been modified.
3. The tables on the local and remote servers must have the same columns and column types, primary keys, and NOT NULL constraints.

3.1.3 Connection Requirements

1. There can be a maximum of 32 replication connections in one Altibase database.
2. The database character sets and the national character sets must be the same on both servers in order for replication to be possible. Which character set is currently in use can be checked by viewing the values of NLS_CHARACTERSET and NLS_NCHAR_CHARACTERSET in the V$NLS_PARAMETERS performance view.

3.1.4 Replication Target Column Constraints

1. When an INSERT transaction is replicated, columns that are not replication targets will be filled with NULL values.
2. When replication target columns and columns that are not replication targets contain unique indexes, the replication object will be successfully created, but cannot be started.

3.1.5 Partitioned Table Constraints

The following conditions must be met in order to successfully replicate partitioned tables.

1. The partitioning method must be the same on both the remote server and the local server.

2. For range or list partitions, the partitioning conditions must be the same.

3. For hash partitions, the number of partitions must be the same.

3.1.6 Restrictions on Using Replication for Data Recovery

In order to use replication to perform data recovery, the following restrictions apply:

1. If both the local server and the remote server shut down abnormally, recovery using replication will not be possible.

2. Conflicting data cannot be recovered.

3. A single table cannot be recovered using two or more replication objects.

4. If transactions that have not been transferred are lost, the data cannot be recovered.

3.1.7 Additional Considerations when Using Replication for Data Recovery

1. If different update operations are performed on the same record on two replicated systems in an Active-Active replication environment, data may be mismatched between the systems.

2. If a network error occurs or replication is stopped according to the setting of the REPLICATION_RECOVERY_MAX_TIME property by the user, data might not be recovered.

3.1.8 Allowable DDL Statements

Normally, DDL statements cannot be executed on replication target tables. However, the following DDL statements can be executed on replication target tables.

- ALTER INDEX SET PERSISTENT = ON/OFF
- ALTER INDEX REBUILD PARTITION
- GRANT OBJECT
- REVOKE OBJECT
- CREATE TRIGGER
- DROP TRIGGER

3.1.8.1 Restrictions

When DDL statements that are allowed for use with replication are executed on tables, those tables are locked. If the Sender thread transfers a replication log at this time, the Receiver thread won't be able to properly implement the log's changes.

3.2 CREATE REPLICATION

Before starting replication, corresponding replication objects must first be created on two servers.

3.2.1 Syntax

```
CREATE [LAZY|EAGER] REPLICATION replication_name
 [AS MASTER|AS SLAVE]
 [OPTIONS option_name [option_name ... ]]
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]
...;
```

3.2.2 Description

Before replication can be performed, a so-called "replication pair", comprising a pair of replication objects between which a connection is established, must be set up.

Replication is conducted on a table-by-table basis. Tables are matched one-to-one.

Up to 32 replication objects can be created on a single server, meaning that it can be connected to a maximum of 32 different remote servers.

When creating a replication object, one of the LAZY and EAGER modes can be selected as the default mode. If the replication mode is not specified for a session, this default mode will be used. If no default mode is specified, replication will be performed in LAZY mode.

- *replication_name*

  This specifies the name of the replication object to be created. The same name must be used on both the local server and the remote server.

- AS MASTER or AS SLAVE

  This specifies whether the server is the Master or the Slave. If not specified, the value specified using the REPLICATION_INSERT_REPLACE or REPLICATION_UPDATE_REPLACE property will be used. When attempting to perform handshaking, the following combinations of values will be successful: 0 with 0, 1 with 2, and 2 with 1. Other combinations will fail. (0 = not set; 1 = Master; 2 = Slave)

- *remote_host_ip*

  This is the IP address of the remote server.

- *remote_host_port_no*

  This is the port number at which the remote server Receiver thread listens. More specifically, this is the port number specified in REPLICATION_PORT_NO in the altibase.properties file on the remote server.

- *user_name*

This is the name of the owner of the table to be replicated.

- *table_name*

  This is the name of the table to be replicated.

- *option_name*

  This is the name of the additional functions (recovery and offline) pertaining to the replication object. The extra features are for use in data recovery and when performing offline replication. For more information, please refer to Extra Features.

3.2.3 Error Codes

Please refer to the *ALTIBASE HDB Error Message Reference*.

3.2.4 Example

Suppose that the IP address and port number of the local server are 192.168.1.60 and 25524, and that the IP address and port number of the remote server are 192.168.1.12 and 35524. To replicate a table called *employees* and one called *departments* between the two servers, the required replication definition would be as follows:

- Local server (IP: 192.168.1.60)

```
iSQL> CREATE REPLICATION rep1
 WITH '192.168.1.12', 35524
 FROM sys.employees TO sys.employees,
 FROM sys.departments TO sys.departments;
Create success.
```

- Remote server (IP: 192.168.1.12)

```
iSQL> CREATE REPLICATION rep1
 WITH '192.168.1.60', 25524
 FROM sys.employees TO sys.employees,
 FROM sys.departments TO sys.departments;
Create success.
```

3.3 Starting, Stopping and Modifying Replication using "ALTER REPLICATION"

3.3.1 Syntax

```
ALTER REPLICATION replication_name
  SYNC [PARALLEL parallel_factor]
      [TABLE user_name.table_name, … , user_name.table_name];

ALTER REPLICATION replication_name
  SYNC ONLY [PARALLEL parallel_factor]
           [TABLE user_name.table_name, … , user_name.table_name];

ALTER REPLICATION replication_name START [RETRY];

ALTER REPLICATION replication_name QUICKSTART [RETRY];

ALTER REPLICATION replication_name STOP;

ALTER REPLICATION replication_name RESET;

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

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

ALTER REPLICATION replication_name FLUSH [ALL] [WAIT timeout_sec];
```

3.3.2 Description

- SYNC

  After all of the records in the table to be replicated have been transmitted from the local server to the remote server, replication starts from the current position in the log. In order to prevent another transaction from changing data in the table on which synchronization is to be performed right at the time of determination of the log from which replication will start after synchronization, the Replication Sender Thread obtains an S Lock on the table on which synchronization is to be performed for a short time before synchronization. Therefore, if a synchronization attempt is made while another transaction is updating data in the table to be synchronized, the Replication Sender Thread will wait for the amount of time specified in the REPLICATION_SYNC_LOCK_TIMEOUT property, and will then start replication at the time at which the change transaction ends. If the change transaction is not completed within the amount of time specified in the REPLICATION_SYNC_LOCK_TIMEOUT property, synchronization will fail. If, during synchronization, records on the local server are found to have the same primary key values as records on the remote server, any conflicts are eliminated according to the rules for conflict resolution.

- TABLE

This specifies the table that is the target for SYNC replication.

- PARALLEL

  Parallel_factor may be omitted, in which case a value of 1 is used by default. The maximum possible value of *parallel_factor* is the number of CPUs * 2. If it is set higher than this number, the maximum number of threads that can be created is still equal to the number of CPUs * 2. If it is set to 0 or a negative number, an error message results.

- SYNC ONLY

  All records in replication target tables are sent from the local server to the remote server. (In this case the Sender thread is not created.) If the same records exist on both the local server and the remote server, sources of conflict are eliminated according to the rules for conflict resolution.

  Because only a single thread is responsible for handling SYNC or SYNC ONLY on disk tables, when some of the tables on which SYNC replication is to be performed are disk tables, setting *parallel_factor* higher than the number of disk tables confers a performance advantage.

- START

  Replication will start from the time point of the most recent replication.

- QUICKSTART

  Replication will start from the current position in the log.

- START/QUICKSTART RETRY

  When starting or quickstarting replication with the RETRY option, even if handshaking fails, a Sender Thread is created on the local server. Afterwards, once handshaking between the local server and the remote server is successful, replication starts.

  iSQL shows a success message even if the first handshake attempt fails. Therefore, the user must check the result of execution of this command by checking the trace logs or the V$REPSENDER performance view.

  When starting replication without the RETRY option, if the first handshake attempt fails, an error is raised and execution is stopped.

- STOP

  This stops replication. If a SYNC task is stopped, the transmission of all data to be replicated to the remote server cannot be guaranteed. If a SYNC replication that is underway is stopped, in order to perform SYNC again, all records must be deleted from all replication target tables, and then the SYNC is performed again.

- RESET

  This command resets replication information (such as the restart SN). It can only be executed while replication is stopped, and can be used instead of executing DROP REPLICATION followed by CREATE REPLICATION.

- DROP TABLE

  This command excludes a table from a replication object. It can only be executed while replication is stopped. Because regular DDL statements cannot be executed on replication target tables, after a table is excluded from a replication object, DDL statements can be executed on the table.

- ADD TABLE

  This command adds a table to a replication object. It can only be executed while replication is stopped.

- FLUSH

  The current session waits for the number of seconds specified by *timeout_sec* so that the replication Sender thread can send logs up to the log at the time at which the FLUSH statement is executed to the other server. If used together with the ALL keyword, the current session waits until the most recent log, rather than the log at the time at which the FLUSH statement is executed, is sent to the other server.

3.3.3 Error Codes

Please refer to the *Error Message Reference*.

3.3.4 Example

Assuming that the name of a replication is *rep1*, replication can be started in one of the following three ways:

- Replication is started after the data on the local server are transferred to the remote server.

```
iSQL> ALTER REPLICATION rep1 SYNC;
Alter success.
```

- Replication is started from the time point at which the replication *rep1* was most recently executed.

```
iSQL> ALTER REPLICATION rep1 START;
Alter success.
```

- Replication is started from the current time point.

```
iSQL> ALTER REPLICATION rep1 QUICKSTART;
Alter success.
```

Use the following commands to check the status of replication after it has started.

```
iSQL> SELECT rep_name, status, net_error_flag, sender_ip, sender_port,
             peer_ip, peer_port
      FROM V$REPSENDER;
REP_NAME                                  STATUS
------------------------------------------------------------------
NET_ERROR_FLAG
-----------------------
SENDER_IP                                                SENDER_PORT
------------------------------------------------------------------------
---
PEER_IP                                                 PEER_PORT
------------------------------------------------------------------------
```

```
---
REP1                                     1
0
192.168.1.33                                                  11477
192.168.1.34                                                  21300
1 row selected.

iSQL> SELECT rep_name, my_ip, my_port, peer_ip, peer_port
      FROM V$REPRECEIVER;
REP_NAME
--------------------------------------------
MY_IP                                                        MY_PORT
------------------------------------------------------------------------
---
PEER_IP                                                      PEER_PORT
------------------------------------------------------------------------
---
REP1
192.168.1.33                                                  21300
192.168.1.34                                                  7988
1 row selected.
```

- Assuming that the name of a replication is *rep1*, use the following command to stop replication.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.
```

- Assuming that the name of a replication is *rep1*, use the following commands to drop a table from a replication object.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.

iSQL> ALTER REPLICATION rep1 DROP TABLE FROM sys.employees TO sys.employ-
ees;
Alter success.
```

- Assuming that the name of a replication is *rep1*, use the following commands to add a table to a replication object.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.

iSQL> ALTER REPLICATION rep1 ADD TABLE FROM sys.employees TO sys.employ-
ees;
Alter success.
```

- If it is desired to check the cumulative time that each Sender replication object has spent waiting for WAIT_NEW_LOG events, execute the following query. This example assumes that the TIMER_THREAD_RESOLUTION property has been set to 1000000 microseconds.

```
select rep_name, avg(WAIT_NEW_LOG)/1000000
from x$repsender_statistics
where wait_new_log > 0
group by rep_name
order by rep_name;
```

- If it is desired to check the cumulative time that each Receiver replication object has spent waiting for INSERT_ROW events, execute the following query. This example assumes that the TIMER_THREAD_RESOLUTION property has been set to 1000000 microseconds.

```
select rep_name, avg(INSERT_ROW)/1000000
from x$repreceiver_statistics
where recv_xlog > 0
group by rep_name
order by rep_name;
```

3.4 DROP REPLICATION

3.4.1 Syntax

```
DROP REPLICATION replication_name;
```

3.4.2 Description

This command is used to remove a replication object.

However, once a replication has been dropped, it cannot be executed using ALTER REPLICATION START. Additionally, in order to drop a replication object, it is first necessary to stop it using ALTER REPLICATION STOP.

3.4.3 Error Codes

Please refer to the *Error Message Reference*.

3.4.4 Example

In the following example, a replication object named *rep1* is removed.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.

iSQL> DROP REPLICATION rep1;
Drop success.
```

If an attempt is made to remove a replication object without first stopping it, the following error message appears.

```
iSQL> DROP REPLICATION rep1;
[ERR-31089 : Replication has already started.]
```

3.5 Executing DDL Statements on Replication Target Tables

3.5.1 Syntax

The DDL statements that ALTIBASE HDB supports for use on replication target tables are as follows:

```
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 PARTITION ...

TRUNCATE TABLE ...

CREATE INDEX ...

DROP INDEX ...
```

3.5.2 Description

ALTIBASE HDB supports the execution of DDL statements on replication target tables. However, the following property settings must first be made.

- The REPLICATION_DDL_ENABLE property must be set to 1.
- The replication session property, set using the ALTER SESSION SET REPLICATION statement, must be set to some value other than NONE.

3.5.3 Restrictions

DDL statements cannot be executed on tables for which the replication recovery option has been specified. To execute DDL statements in such a case, drop the tables from the replication object and execute the DDL statements. Futhermore, DDL statements cannot be executed while replication is running in EAGER mode. To execute DDL statements in such a case, stop replication, execute the DDL statements, and start replication again. The restrictions that govern the use of particular DDL statements are as follows:

- ALTER TABLE *table_name* ADD COLUMN
  - A column having a `NOT NULL` cannot be added.
  - A unique index cannot be added.
  - A foreign key cannot be added.
- ALTER TABLE *table_name* DROP COLUMN

- A column having a `NOT NULL` constraint cannot be added.
- A unique index cannot be added.
- The primary key cannot be deleted.

- CREATE INDEX
  - This is supported only for indexes that are not unique.
- DROP INDEX
  - This is supported only for indexes that are not unique.

3.5.4 Example

Supposing that the name of a replication target table is *t1*, DDL statements can be executed on the replication target table as follows.

- Execution of the `TRUNCATE TABLE` statement.

(SYS User)

```
iSQL> ALTER SYSTEM SET REPLICATION_DDL_ENABLE = 1;
Alter success.
```

(Table Owner)

```
iSQL> ALTER SESSION SET REPLICATION = DEFAULT;
Alter success.

iSQL> TRUNCATE TABLE t1;
Truncate success.
```

(SYS User)

```
iSQL> ALTER SYSTEM SET REPLICATION_DDL_ENABLE = 0;
Alter success.
```

3.6 Extra Features

ALTIBASE HDB provides the following extra replication features:

- Recovery Option
- Offline Option

3.6.1 Recovery Option

3.6.1.1 Syntax

```
ALTER REPLICATION replication_name SET RECOVERY {ENABLE|DISABLE};
```

3.6.1.2 Description

One of the extra replication features that ALTIBASE HDB supports is the recovery option. If the OPTIONS value is set to 1 in the SYS_REPLICATIONS_ meta table, the recovery option is used, whereas if the OPTIONS value is set to 0, the recovery option is not used. However, the recovery option cannot be changed while replication is active. If the recovery option is not used, all of the recovery-related information maintained in the system is cleared.

3.6.1.3 Restriction

The recovery option cannot be used at the same time as the offline option.

3.6.1.4 Example

Assuming that the name of a replication object is *rep1*, the replication recovery option is used as follows:

- To use the replication recovery option:

```
iSQL> ALTER REPLICATION rep1 SET RECOVERY ENABLE;
Alter success.
```

- To stop using the replication recovery option:

```
iSQL> ALTER REPLICATION rep1 SET RECOVERY DISABLE;
Alter success.
```

3.6.2 Offline Option

3.6.2.1 Syntax

```
ALTER REPLICATION replication_name
  SET OFFLINE ENABLE WITH 'log_dir_1', 'log_dir_2', ..., 'log_dir_n';

ALTER REPLICATION replication_name SET OFFLINE DISABLE;
```

```
ALTER REPLICATION replication_name START WITH OFFLINE;
```

3.6.2.2 Description

One of the other extra replication features provided with ALTIBASE HDB is the offline option. In an Active-Passive replication environment, when a server providing service (the “Active“ server) develops a fault, the logs cannot be sent to the remote (“Standby“) server. The use of offline replication allows the logs that could not be sent to the Standby Server before the fault occurred to be accessed by and implemented in the Standby Server afterwards. If the Standby Server directly accesses the log files on the Active Server by copying the files via FTP or using a shared disk file system, a network file system, etc, the logs that could not be sent can be processed using the OFFLINE option.

If the value of OPTIONS in the SYS_REPLICATIONS_ meta table is set to 2, the offline option is used, whereas if it is set to 0, the offline option is not used.

- log_dir_n

  This enables the Standby Server to access the log files directly by specifying the log path on the Active Server.

- START WITH OFFLINE

  This allows replication to take place using the specified offline path.

3.6.2.3 Offline Option Restrictions

- This option cannot be used while replication is running in EAGER mode.
- The offline option cannot be used at the same time as the recovery option.
- At the moment that offline replication starts, any replication Receiver thread having the same *replication_name* must be in a stopped state. If such a thread is still running, offline replication will terminate.
- If the log file directory on the Active Server cannot be accessed due to a disk error, offline replication will fail.
- The size of the log files on the Active and Standby Servers must be the same. Before the offline option is used, it must be ensured that the size of the log files is the same as the size that was specified at the time that the database was created.
- If the user changes log files arbitrarily (i.e. renames or deletes them, or copies log files from another system), abnormal shutdown or some other problem may occur.
- The Standby Server should not be restarted before starting offline replication, because the information used to analyze the logs that could not be received will disappear when starting up the Standby Server.
- The attempt to start a replication for which the offline option has been enabled or a replication that was created with the offline option will fail if the information about the SM version, OS, the number of OS bits (i.e. 32 or 64), the size of the log files or the number of LFGs differs between the two database servers.

3.6.2.4 Example

Assuming that the name of a replication object is *rep1* and that the path of Active Server logs is active_server/altibase_home/logs, the offline option is used as follows:

- Setting the offline option when creating a replication object:

```
iSQL> CREATE REPLICATION rep1 OPTIONS OFFLINE 'actiive_server/
altibase_home/logs'
WITH '127.0.0.1',20300 FROM SYS.A TO SYS.B;
```

- Setting the offline option for an existing replication object:

```
iSQL> ALTER REPLICATION rep1 SET OFFLINE ENABLE WITH 'active_server/
altibase_home/logs';
```

- Executing offline replication using the specified path:

```
iSQL> ALTER REPLICATION rep1 START WITH OFFLINE;
```

- Specifying that the offline option is not to be used:

```
iSQL> ALTER REPLICATION rep1 SET OFFLINE DISABLE;
```

3.7 Replication in a Multiple IP Network Environment

Replication is supported in a multiple IP network environment. In other words, it is possible to perform replication between two hosts having two or more physical network connections therebetween.

3.7.1 Syntax

```
CREATE REPLICATION replication_name AS {MASTER|SLAVE}
  WITH 'remotehostip', remoteportno  'remotehostip', remoteportno …
  FROM user.localtableA TO user.remotetableA,
  FROM user.localtableB TO user.remotetableB,
  …
  FROM user.localtableC TO user.remotetableC;

ALTER REPLICATION replication_name
  ADD HOST 'remotehostip', remoteportno;

ALTER REPLICATION replication_name
 DROP HOST 'remotehostip', remoteportno;

ALTER REPLICATION replication_name
  SET HOST 'remotehostip', remoteportno;
```

3.7.2 Description

In order to ensure high system performance and quickly overcome faults, systems can have multiple physical IP addresses assigned to them when a replication object is created. In such an environment, the Sender thread uses the first IP address to access peers and perform replication tasks when replication starts, but if a problem occurs while this task is underway, the Sender thread stops using this connection, connects using another IP address, and tries again.

- CREATE REPLICATION

  The name of the replication object is first specified, and then in the WITH clause, the IP addresses and reception ports of multiple remote servers are specified, with commas between each IP address and port, and with spaces between address/port pairs defining each host. The owner and name of the target table(s) on the local server are specified in the FROM clause and the owner and name of the corresponding target table(s) on the remote server are specified in the TO clause, with commas between multiple table specifications.

- ALTER REPLICATION (ADD HOST)

  This adds a host. A host can be added to a replication object after the replication object has been stopped. When ADD HOST is executed, before the Sender thread actually adds the host, the connection must be re-established using the IP address that was previously being used.

- ALTER REPLICATION (DROP HOST)

  This drops a host. A host can be dropped from a replication object after the replication object has been stopped. When DROP HOST is executed, the Sender thread attempts to reconnect using the very first IP address.

- ALTER REPLICATION (SET HOST)

  This means setting a particular host as the current host. The current host can be specified after the replication object has been stopped. After execution, the Sender thread attempts to connect using the currently designated IP address.

3.7.3 Examples

In the following double-IP network environment, a replication object having a table called *employees* and one called *departments* as its target objects is created, and then replication in Active-Standby mode is executed on the local server (IP: 192.168.1.51, PORT NO: 30570) and the remote server ('IP: 192.168.1.154, PORT NO: 30570', 'IP: 192.168.2.154, PORT NO: 30570').

- On the remote (standby) server:

```
iSQL> CREATE REPLICATION rep1
WITH '192.168.1.51',30570
FROM sys.employees TO sys.employees,
FROM sys.departments TO sys.departments;
Create success. <- The replication object is created on the remote server.
```

- On the local (active) server:

```
iSQL> CREATE REPLICATION rep1
WITH '192.168.1.154',30570 '192.168.2.154',30570
FROM sys.employees TO sys.employees,
FROM sys.departments TO sys.departments;
Create success.<- The replication object is created on the local server.

iSQL> SELECT * FROM system_.sys_replications_; <- The meta table enables the user to
view the number of registered hosts, the number of replication target tables, and other related informa-
tion.
REPLICATION_NAME        LAST_USED_HOST_NO    HOST_COUNT
------------------------------------------------------------------------
IS_STARTED  XSN         ITEM_COUNT  CONFLICT_RESOLUTION REPL_MODE
------------------------------------------------------------------------
ROLE        OPTIONS     INVALID_RECOVERY REMOTE_FAULT_DETECT_TIME
-----------------------------------------------------------------------
REP1                    2                    2
0           -1          2           0                   0
0           0           0
1 row selected.

iSQL> SELECT * FROM system_.sys_repl_hosts_; <- The meta table enables the user to
view the remote server-related information.
HOST_NO     REPLICATION_NAME
---------------------------------------------------------
HOST_IP                                                      PORT_NO
------------------------------------------------------------------------
2           REP1
192.168.1.154                                                 30570
3           REP1
192.168.2.154                                                 30570
2 rows selected.

iSQL> ALTER REPLICATION rep1 START; <- Replication starts
Alter success.

iSQL> SELECT rep_name, status, net_error_flag, sender_ip, sender_port,
             peer_ip, peer_port
FROM V$REPSENDER;
```

```
REP_NAME                                   STATUS
-----------------------------------------------------------------
NET_ERROR_FLAG
-----------------------
SENDER_IP                                                   SENDER_PORT
-----------------------------------------------------------------------
PEER_IP                                                   PEER_PORT
-----------------------------------------------------------------------
REP1                                       1
0
192.168.1.51                                                13718
192.168.1.154                                               30570
1 row selected.
```
<- The status of replication is checked after replication starts. The Sender thread connects to the peer using the first IP and PORT.

```
!!!!!!!!!!!!! Network line disconnection !!!!!!!!!!!!!!!

iSQL> SELECT rep_name, status, net_error_flag, sender_ip, sender_port,
             peer_ip, peer_port
FROM V$REPSENDER;
REP_NAME                                   STATUS
-----------------------------------------------------------------
NET_ERROR_FLAG
-----------------------
SENDER_IP                                                   SENDER_PORT
----------------------------------------------------------------------
PEER_IP                                                   PEER_PORT
----------------------------------------------------------------------
REP1                                       1
0
192.168.1.51                                                40009
192.168.2.154                                               30570
1 row selected.
```
<- The status of replication is checked after network failure occurs. This verifies reconnection using the second IP and PORT.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.
```
<- Replication is stopped

```
iSQL> ALTER REPLICATION rep1 START;
Alter success.
```
<- Replication starts

```
iSQL> SELECT rep_name, status, net_error_flag, sender_ip, sender_port,
             peer_ip, peer_port
FROM V$REPSENDER;
REP_NAME                                   STATUS
-----------------------------------------------------------------
NET_ERROR_FLAG
-----------------------
SENDER_IP                                                   SENDER_PORT
----------------------------------------------------------------------
PEER_IP                                                   PEER_PORT
----------------------------------------------------------------------
REP1                                       1
0
192.168.1.51                                                64351
192.168.2.154                                               30570
1 row selected.
```
<- When replication is started again after having been stopped, it can be verified to have been reconnected to the same IP and PORT to which it was connected before being stopped.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.
```
<- Replication is stopped

```
iSQL> ALTER REPLICATION rep1 ADD HOST '192.168.3.154',30570;
Alter success.
```
<- Add host: Can be executed after replication.

```
iSQL> ALTER REPLICATION rep1 DROP HOST '192.168.3.154',30570;
Alter success.  <- remove host: Can be executed after replication.

iSQL> ALTER REPLICATION rep1 SET HOST '192.168.1.154',30570;
Alter success.<- Designate the host: Can be executed after replication.

iSQL> ALTER REPLICATION rep1 START;
Alter success.<- Replication is restarted after setting the new host. The replication operation first
attempts to connect using the currently designated IP and PORT.

iSQL> SELECT rep_name, status, net_error_flag, sender_ip, sender_port,
             peer_ip, peer_port
FROM V$REPSENDER;
REP_NAME                                  STATUS
------------------------------------------------------------------
NET_ERROR_FLAG
-----------------------
SENDER_IP                                                     SENDER_PORT
-----------------------------------------------------------------------
PEER_IP                                                      PEER_PORT
-----------------------------------------------------------------------
REP1                                      1
0
192.168.1.51                                                  11477
192.168.1.154                                                 30570
1 row selected.  <- Connection to the peer using the newly designated IP 192.168.1.154 and PORT
number 30570 can be confirmed.
```

- The following messages are written to altibase_rp.log during execution of the above-mentioned example.
- By enabling the HeartBeat Trace log, it is possible to check whether the HeartBeat Thread was active.

```
iSQL> ALTER SYSTEM SET RP_MSGLOG_FLAG = 7;  <- Default value is 6
```

- The following message is written to the log file after a replication object is created. Whether the corresponding host has failed is checked at intervals corresponding to REPLICATION_HBT_DETECT_TIME, which in this case has been set to 3 seconds.

```
[2010/10/28 15:49:44] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
[2010/10/28 15:49:47] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
[2010/10/28 15:49:50] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
```

- The following message can be seen when replication starts. Connection to the peer using the first IP and PORT can be verified.

```
[2010/10/28 15:50:44] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
[2010/10/28 15:50:44] [Thread-1649092928] [Level-1]
[HBT] Host status info.
      [192.168.1.34:21300] Ref=1 Mode=0 mFault=No Fault
[2010/10/28 15:50:47] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
[2010/10/28 15:50:47] [Thread-1649092928] [Level-1]
[HBT] Host status info.
      [192.168.1.34:21300] Ref=1 Mode=0 mFault=No Fault
[2010/10/28 15:50:50] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
[2010/10/28 15:50:50] [Thread-1649092928] [Level-1]
```

```
[HBT] Host status info.
      [192.168.1.34:21300] Ref=1 Mode=0 mFault=No Fault
```

- If the REPLICATION_HBT_DETECT_HIGHWATER_MARK, which is one of the ALTIBASE HDB properties, is set to 10 after the network line has been disconnected, the WaterMark value can be confirmed to have been changed from 1 to 10. Thus, the HeartBeat thread would determine that failure has occurred after not having received a response after 10 attempts, and an attempt would be made to connect to the next host using the next IP and port number.

```
[2010/10/28 16:02:36] [Thread-1638603072] [Level-0]
[Sender] getNextLastUsedHostNo: from 192.168.1.34:21300 to
192.168.1.35:21300
...
[2010/10/28 16:04:17] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
[2010/10/28 16:04:17] [Thread-1649092928] [Level-1]
[HBT] Host status info.
      [192.168.1.34:21300] Ref=1 Mode=0 mFault=No Fault
```

- The following message will be output when replication stops:

```
[2010/10/28 15:58:59] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
[2010/10/28 15:59:02] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
[2010/10/28 15:59:05] [Thread-1649092928] [Level-1]
[HBT] == Network Fault Detection Proceeding ==
```

3.8 Properties

To use replication, the ALTIBASE HDB properties file should be modified to suit the purposes of the user. The following properties are described in the *Altibase HDB Starting User's Manual*.

- REPLICATION_ACK_XLOG_COUNT
- REPLICATION_CONNECT_RECEIVE_TIMEOUT
- REPLICATION_CONNECT_TIMEOUT
- REPLICATION_DDL_ENABLE
- REPLICATION_HBT_DETECT_HIGHWATER_MARK
- REPLICATION_HBT_DETECT_TIME
- REPLICATION_INSERT_REPLACE
- REPLICATION_KEEP_ALIVE_CNT
- REPLICATION_LOCK_TIMEOUT
- REPLICATION_LOG_BUFFER_SIZE
- REPLICATION_MAX_LOGFILE
- REPLICATION_POOL_ELEMENT_COUNT
- REPLICATION_POOL_ELEMENT_SIZE
- REPLICATION_PORT_NO
- REPLICATION_PREFETCH_LOGFILE_COUNT
- REPLICATION_RECEIVE_TIMEOUT
- REPLICATION_RECOVERY_MAX_LOGFILE
- REPLICATION_RECOVERY_MAX_TIME
- REPLICATION_SENDER_AUTO_START
- REPLICATION_SENDER_SLEEP_TIME
- REPLICATION_SENDER_SLEEP_TIMEOUT
- REPLICATION_SERVICE_WAIT_MAX_LIMIT
- REPLICATION_SYNC_LOCK_TIMEOUT
- REPLICATION_SYNC_LOG
- REPLICATION_SYNC_TUPLE_COUNT
- REPLICATION_TIMESTAMP_RESOLUTION

- REPLICATION_UPDATE_REPLACE
- REPLICATION_EAGER_PARALLEL_FACTOR
- REPLICATION_COMMIT_WRITE_WAIT_MODE
- REPLICATION_SERVER_FAILBACK_MAX_TIME
- REPLICATION_FAILBACK_PK_QUEUE_TIMEOUT

3.8 Properties

4 Fail-Over

The Fail-Over feature is provided so that a fault that occurs while a database is providing service can be overcome and service can continue to be provided as though no fault had occurred. This chapter explains the Fail-Over feature that is provided with ALTIBASE HDB, and how to use it.

4.1 Fail-Over Overview

4.1.1 Concept

“Fail-Over” refers to the ability to overcome a fault that occurs while a database is providing service, so that service can continue to be provided as though no fault had occurred.

The kinds of faults that can occur include the case in which the DBMS server hardware itself develops a fault, the case in which the server’s network connection is interrupted, and the case in which a software error causes the DBMS to shut down abnormally. When any of the above kinds of fault occurs, Fail-Over makes it possible to connect to another server, so that service can be provided without interruption, and so that client applications are never aware that a fault has occurred.

There are two kinds of Fail-Over, distinguished from each other according to the time point at which the existence of a fault becomes known:

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

CTF refers to the case where the fault is noted at the time of connection to the DBMS, and connection is made to a DBMS on another available node rather than to the DBMS suffering from the fault, so that service can continue to be provided.

In the case of STF, in contrast, 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. Therefore, tasks currently being executed on the DBMS in which the fault occurred may need to be executed again.

With this kind of Fail-Over, in order to have confidence in the results of a task, the databases on the DBMS in which the fault occurred and the DBMS that is available to provide service must be guaranteed to be in exactly the same state and to 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 Standby 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 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.

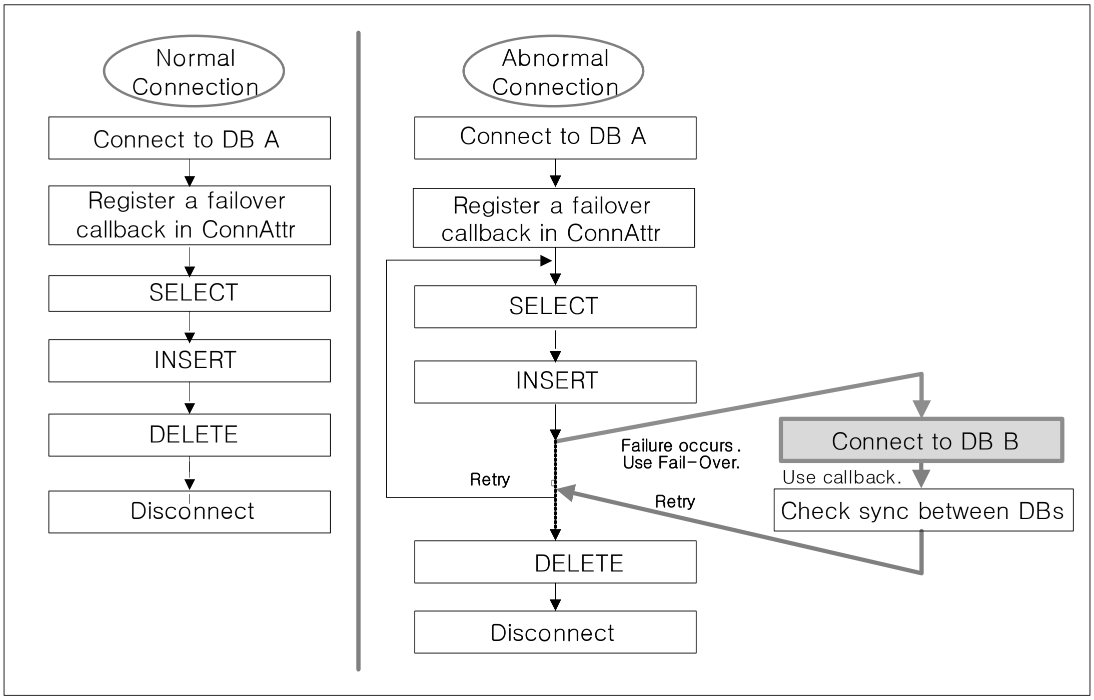
The three kinds of Fail-Over-related tasks that must be executed by the client application are summarized as follows:

- 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

4.1.2 Process

The Fail-Over registration and handling process is as shown in the following figure.

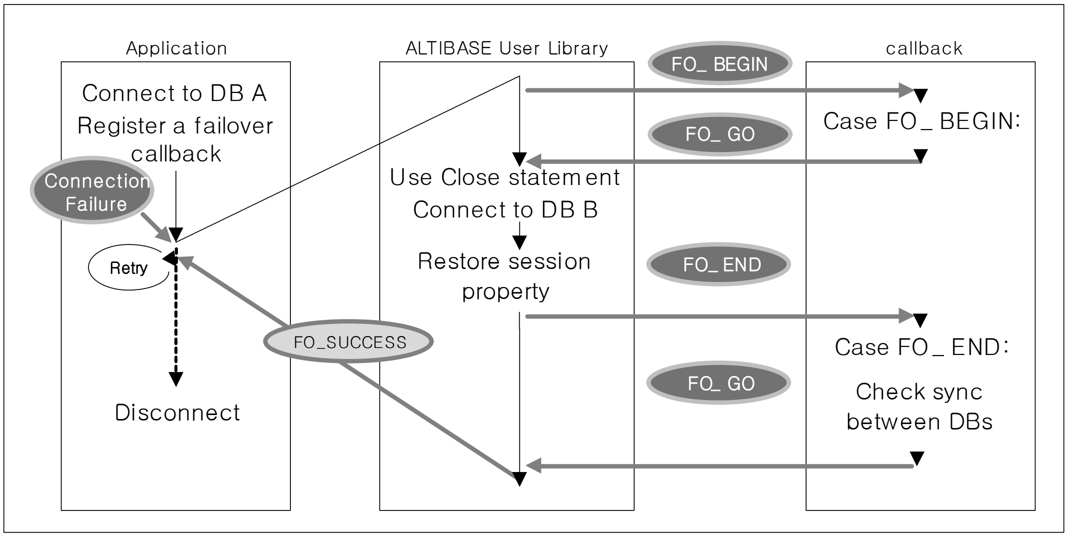
Figure 4-1 Fail-Over Registration and Handling Process



Fail-Over Callback must be registered by the user, and, once registered, during the Fail-Over process the ALTIBASE HDB User Library (for example, the JDBC and CLI libraries) communicates with client applications, as shown in the picture above.

If Fail-Over Callback is not registered, Fail-Over takes place without communication with the client application, and a trace log of the steps shown above is kept. In a replicated Altibase database environment, the use of callback is strongly recommended, so that Fail-Over Validation can be conducted.

Figure 4-2 Fail-Over Process



1. After connecting to the database, the user registers Fail-Over Callback in the connection attributes.

2. The business logic is conducted in the client application. While the client application is running, if it receives an error message about a fault occurring in the DBMS hardware (including a network error), it calls the ALTIBASE HDB User Library so that Fail-Over can be conducted.

3. This client library sends a Fail-Over Start Event (FO_BEGIN) to the registered Fail-Over Callback. Fail-Over Callback returns information about whether Fail-Over will continue to progress.

4. If Fail-Over Callback determines that the Fail-Over process should continue (FO_GO), executed SQL statements are closed, an available server is located, and the ALTIBASE HDB User Library connects and logs in to that database. Additionally, the properties of the previous session (autocommit mode, optimization settings, XA connection settings, etc.) are restored on the new server.

5. When step number 4 is complete, Fail-Over Callback sends an event indicating that the Fail-Over process has been completed successfully (FO_END).

6. Fail-Over Callback executes a query to ensure that the databases match (Fail-Over Validation). In a replicated database environment, it is essential to ensure that the databases match.

4.2 Using Fail-Over

4.2.1 Registering Connection Properties

Once the Fail-Over connection properties have been registered, when a fault occurs, ALTIBASE HDB detects this and internally conducts the Fail-Over tasks according to the expressly specified connection properties.

The properties can be specified in the following two ways:

- by specifying the Connection String when calling the API's connection function
- by specifying connection properties in the appropriate ALTIBASE HDB settings file (altibase_cli.ini or odbc.ini [WinODBC])

4.2.1.1 Specifying the Connection String in a Client Application

When the connection function is executed in the client application, the following connection strings can be specified:

- JDBC

```
Jdbc:Altibase://192.168.3.51:20300/mydb?AlternateServ-
ers=(192.168.3.54:20300,192.168.3.53:20300)&ConnectionRetryCount=3&Con-
nectionRetryDelay=3&LoadBalance=off&SessionFailOver=on;
```

- ODBC, Embedded SQL

```
DSN=192.168.3.51;UID=altibase;PWD=altibase;PORT_NO=20300;AlternateServer
s=(192.168.3.54:20300,192.168.3.53:20300);ConnectionRetryCount=3;Connec-
tionRetryDelay=5;LoadBalance=on;SessionFailOver=on;
```

`AlternateServer` indicates servers to which connection can be made in the event of a fault, and is expressed in the form (IP Address1:Port1, IP Address2:Port2,...).

`ConnectionRetryCount` indicates the number of times to repeatedly attempt to connect to an available server in the event of a connection failure.

`ConnectionRetryDelay` indicates the amount of time to wait between connection attempts in the event of a connection failure.

When `LoadBalance` is set to ON, the first connection attempt will be made to a server that is randomly selected from among the group comprising the default server and the alternate servers. When it is set to OFF, the first connection attempt is made to the default server, and if that fails, subsequent connection attempts are made to the server(s) specified in `AlternateServer`.

`SessionFailOver` indicates whether STF (Service Time Fail-Over) is to be conducted.

4.2.1.2 Specifying the Connection Properties in the Settings File

The Fail-Over connection settings can be specified in the Data Source portion of the altibase_cli.ini file, which is located in the $ALTIBASE_HOME/conf directory, the $HOME directory, or the current directory of the relevant client application, and the DataSource name is specified in the Connection

String of the connection function.

```
[MyDataSource1]
Server=192.168.3.51
Port=20300
User=altibase
Password=altibase
DataBase = mydb
AlternateServers=(192.168.3.54:20300,192.168.3.53:20300)
ConnectionRetryCount=3
ConnectionRetryDelay=5
LoadBalance = on
SessionFailOver = off
```

The Connection String of the client application's connection function appears as shown below, depending on the connection interface used by the client application.

- JDBC

  The data source name is specified as part of the Connection URL as follows:

  ```
  Jdbc:Altibase://MyDataSource1//
  ```

- ODBC, Embedded SQL

  The data source name is specified in the DSN properties as follows:

  ```
  DSN=MyDataSource
  ```

Settings are made in the odbc.ini file in the same way that they are made in the altibase_cli.ini file.

4.2.2 Checking Whether Fail-Over Succeeded

Whether CTF (Connection Time Fail-Over) was successful can be quickly and easily determined merely by checking whether it is possible to connect to the database. In contrast, determining whether STF (Service Time Fail-Over) was successful involves checking for exceptions and errors.

For example, when using JDBC, a SQLException is caught, and the SQLException's getSQLState() method is used to check the value of SQLStates.status. If this value is ES_08FO01, Fail-Over is determined to have been successful.

When using a CLI or ODBC, if the result of SQLPrepare, SQLExecute, SQLFetch or the like is an error, rather than SQL_SUCCESS, a statement handle is handed over to the `SQLGetDiagRec` function, and if the native error code that is returned in the 5th argument of this function has a diagnostic record equal to ALTIBASE_FAILOVER_SUCCESS, STF (Service Time Fail-Over) can be determined to have succeeded.

When using Embedded SQL, after executing the EXEC SQL command, if sqlca.sqlcode is not SQL_SUCCESS but ALTIBASE_FAILOVER_SUCCESS, this means that STF (Service Time Fail-Over) was successful.

The actual method of determining whether Fail-Over has succeeded varies according to the type of client application, as will be explained below.

4.2.3 Writing Fail-Over Callback Functions

It is necessary to write a callback function to determine whether databases match when Fail-Over is executed. The method of writing Fail-Over Callback functions varies depending on the type of client application, but the basic structure is the same, and is as follows:

- define data structures related to Fail-Over
- write Fail-Over Callback function bodies for handling Fail-Over-related events
- write code to determine whether Fail-Over was successful

Either Fail-Over events are defined in the data structure definition, or else a defined interface (header file) is included in the data structure definition.

Various tasks must be conducted in response to Fail-Over-related events such as the start or completion of Fail-Over. Code for performing these tasks, including for example the task of checking whether the contents of databases match, is located in the callback function body.

Determining that Fail-Over has succeeded comprises the successful completion of Fail-Over and the successful execution of a Fail-Over callback function, and means that service that was suspended due to a fault can continue to be provided.

The actual method of writing callback functions is described below for various client application environments.

4.3 JDBC

4.3.1 Fail-Over Callback Interface

```
public interface ABFailOverCallback
{
int FO_BEGIN = 0;
int FO_END   = 1;
int FO_ABORT = 2;
int FO_GO    = 3;
int FO_QUIT  = 4;
int failOverCallback(Connection aConnection,
                     Object     aAppContext,
                     int        aFailOverEvent);
};
```

The meaning of the values is as follows:

FO_BEGIN

FailOverCallback is notified of the start of STF (Service Time FailOver).

FO_END

FailOverCallback is notified of the success of STF.

FO_ABORT

FailOverCallback is notified of the failure of STF.

FO_GO

FailOverCallback sends this to JDBC so that STF can advance to the next step.

FO_QUIT

This is used to notify JDBC of the failure of FailOverCallback.

aAppContext

This includes information about any objects that the user intends to save. If there are no objects to be saved, this is set to NULL.

4.3.2 Writing Fail-Over Callback Functions

The MyFailOverCallback class, which implements the ABFailOverCallback Interface, must be written.

The tasks to be conducted in response to the FO_BEGIN and FO_END events, which are defined in the callback interface, must be handled by this class. That is to say, the required tasks for each of the Fail-Over events are described here.

For example, when the FO_BEGIN event occurs, code for handling tasks that are required before Fail-Over starts is provided, and when the FO_END event occurs, code for handling tasks that are required after Fail-Over ends and before service resumes is provided. One concrete example is the code that is used to check whether the data are consistent between available databases when the

FO_END event occurs.

```
public class MyFailOverCallback implements ABFailOverCallback
{
    public int failOverCallback(Connection aConnection,
                                Object    aAppContext,
                                int        aFailOverEvent)
    {
        Statement sStmt  = null;
        ResultSet sRes  =  null;

        switch (aFailOverEvent)
        {
            case ABFailOverCallback.FO_BEGIN:
                System.out.println("FailOver Started .... ");
                break;

            case ABFailOverCallback.FO_END:
                try
                {
                    sStmt = aConnection.createStatement();
                }
                catch( SQLException ex1 )
                {
                    try
                    {
                        sStmt.close();
                    }
                    catch( SQLException ex3 )
                    {
                    }
                    return ABFailOverCallback.FO_QUIT;
                } //catch  SQLException ex1

                try
                {
                    sRes = sStmt.executeQuery("select 1 from dual");
                    while(sRes.next())
                    {
                        if(sRes.getInt(1) == 1 )
                        {
                            break;
                        }
                    }//while;
                }
                catch ( SQLException ex2 )
                {
                    try
                    {
                        sStmt.close();
                    }
                    catch( SQLException ex3 )
                    {
                    }
                    return ABFailOverCallback.FO_QUIT;
                }//catch
                break;
        }//switch
        return ABFailOverCallback.FO_GO;
    }
}
```

Furthermore, the MyFailOverCallback class defined above is used to create a callback object.

```
MyFailOverCallback sMyFailOverCallback = new MyFailOverCallback();
Properties sProp = new Properties();
String sURL =
"jdbc:Altibase://192.168.3.51:20300+"/mydb?connectionRetryCount=3&
connectionRetryDelay=10&sessionFailOver=on&loadBalance=off";
```

The created callback object is registered with the connection object.

```
((ABConnection)sCon).registerFailOverCallback(sMyFailOverCallback,null);
```

4.3.3 Checking Whether Fail-Over Succeeded

Checking whether Fail-Over, particularly STF (Service Time Fail-Over), was successful is conducted using SQLException. An SQLException is caught, and the SQLException's getSQLState() method is used to check the value of SQLStates.status. If this value is ES_08FO01, Fail-Over is determined to have been successful.

The following example demonstrates how to check whether Fail-Over was successful.

```
while(true)
{
  try
  {
    sRes = sStmt.executeQuery("SELECT C1 FROM T1");
    while( sRes.next() )
    {
      System.out.println( "VALUE : " + sRes.getString(1) );
    }//while
    break;
  }
  catch ( SQLException e )
  {
    if(e.getSQLState().equals(SQLStates.status[SQLStates.ES_08FO01]) == true)
    {
      continue;
    }
    System.out.println( "EXCEPTION : " + e.getMessage() );
    break;
  }
}
```

4.3.4 Sending Fail-Over Connection Settings to WAS

The Fail-Over property settings are added to the URL portion as follows:

```
"jdbc:Altibase://192.168.3.51:20300+"/mydb?connectionRetryCount=3&connec-
tionRetryDelay=10&sessionFailOver=on&loadBalance=off";
```

4.3.5 Example

When the callback functions defined above are used, client applications are authored as seen below.

Please refer to the following example, which is included with the ALTIBASE HDB package and should have been installed in $ALTIBASE_HOME/sample/JDBC/Fail-Over/FailOverCallbackSample.java.

When Fail-Over is completed, whether Fail-Over was successful is checked using SQLStates. The value of the element at index SQLStates.ES_08FO01 in the SQLStates.status array indicates that Fail-Over was successful, and that the client application can resume its tasks and service can be provided again.

```
class FailOverCallbackSample
{
 public static void main(String args[]) throws Exception
 {
 //-----------------------------------------------------
 // Initialization
 //-----------------------------------------------------
 // AlternateServers is the available node property.
 String sURL = "jdbc:Altibase://127.0.0.1:" +
args[0]+"/mydb?AlternateServers=(128.1.3.53:20300,128.1.3.52:20301)&
ConnectionRetryCount=100&ConnectionRetryDelay=100&SessionFailOver=on&
LoadBalance=off";

 try
 {
 Class.forName("Altibase.jdbc.driver.AltibaseDriver");
 }
 catch ( Exception e )
 {
 System.err.println("Can't register Altibase Driver\n");
 return;
 }
 //-----------------------------------------------------
 // Test Body
 //-----------------------------------------------------
 //-----------------------
 // Preparation
 //-----------------------
 Properties sProp = new Properties();
 Connection sCon;
 PreparedStatement sStmt = null;
 ResultSet sRes = null ;
 sProp.put("user", "SYS");
 sProp.put("password", "MANAGER");

 MyFailOverCallback sMyFailOverCallback = new MyFailOverCallback();
 sCon = DriverManager.getConnection(sURL, sProp);
 //FailOverCallback is registered.
 ((ABConnection)sCon).registerFailOverCallback(sMyFailOverCallback, null);
 // Programs must be written in the following form in order to support Session
Fail-Over.
 /*
 while (true)
 {
 try
 {

 }
 catch( SQLException e)
 {
 //Fail-Over occurs.
 if(e.getSQLState().equals(SQLStates.status[SQLStates.ES_08FO01]) == true)
 {
 continue;
 }
 System.out.println( "EXCEPTION : " + e.getMessage() );
 break;
 }
 break;
 } // while
```

```
*/

while(true)
{
try
{
sStmt = sCon.prepareStatement("SELECT C1 FROM T2 ORDER BY C1");
sRes = sStmt.executeQuery();
while( sRes.next() )
{
System.out.println( "VALUE : " + sRes.getString(1) );
}//while
}

catch ( SQLException e )
{
//FailOver occurs.
if(e.getSQLState().equals(SQLStates.status[SQLStates.ES_08FO01]) == true)
{
continue;
}
System.out.println( "EXCEPTION : " + e.getMessage() );
break;
}
break;
}
sRes.close();
//---------------------------------------------------
// Finalize
//---------------------------------------------------

sStmt.close();
sCon.close();
}
}
```

4.4 SQL CLI

In this section, the structure of sqlcli.h and the Fail-Over related constants that are declared therein will be examined, and how to register Fail-Over Callback will be explained with reference to an example.

4.4.1 Related Data Structures

The prototype of the Fail-Over callback function, used for communication between the client application and the CLI library during STF (Service Time Fail-Over), is shown below.

```
typedef  SQLUINTEGER  SQL_API (*SQLFailOverCallbackFunc)
           (SQLHDBC          aDBC,
            void            *aAppContext,
            SQLUINTEGER      aFailOverEvent);
```

aDBC is the SQLHDBC created by the client application using SQLAllocHandle.

aAppContext is a pointer, sent to the CLI library at the time of registration of FailOverCallbackContext, pointing to an object that the user wishes to save. When Fail-Over callback is called at the time of STF (Service Time FailOver), it is sent again to Fail-Over callback.

aFailOverEvent can be set to the following values, which have the meanings described below.

ALTIBASE_FO_BEGIN: 0

Fail-Over callback is notified of the start of STF (Service Time FailOver).

ALTIBASE_FO_END: 1

Fail-Over callback is notified of the success of STF (Service Time FailOver).

ALTIBASE_FO_ABORT: 2

Fail-Over callback is notified of the failure of STF (Service Time FailOver).

ALTIBASE_FO_GO: 3

Fail-Over callback sends aFailOverEvent to the CLI library so that STF can advance to the next step.

ALTIBASE_FO_QUIT: 4

Fail-Over callback sends aFailOverEvent to the CLI library to prevent STF from advancing to the next step.

The structure of SQLFailOverCallbackContext is as follows.

```
typedef struct SQLFailOverCallbackContext
{
 SQLHDBC mDBC;
 void *mAppContext;
 SQLFailOverCallbackFunc mFailOverCallbackFunc;
}SQLFailOverCallbackContext;
```

In the case of CLI, mDBC can be set to NULL.

mAppContext includes information about any objects that the user intends to save. If there are no objects to be saved, this is set to NULL.

mFailOverCallbackFunc is the name of the user-defined FailOverCallback function.

4.4.2 Registering Fail-Over

As can be seen below, the process of Fail-Over registration involves the creation of a FailOverCallbackContext object, and after connection to the database is successful, FailOverCallbackContext is populated with values.

The following is an example of Fail-Over registration.

```
SQLFailOverCallbackContext sFailOverCallbackContext;
...... <<some code omitted here>>
 /* connect to server */
sRetCode = SQLDriverConnect(sDbc, NULL,
SQLCHAR*)"DSN=127.0.0.1;UID=unclee;PWD=unclee;PORT_NO=20300;
AlternateServers=(192.168.3.54:20300,192.168.3.53:20300);ConnectionRetry-
Count=3;
ConnectionRetryDelay=5;LoadBalance=on;SessionFailOver=on;"),
SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
sFailOverCallbackContext.mDBC = NULL;
sFailOverCallbackContext.mAppContext = NULL;
sFailOverCallbackContext.mFailOverCallbackFunc = myFailOverCallback;
sRetCode = SQLSetConnectAttr(sDbc,ALTIBASE_FAILOVER_CALLBACK,
 (SQLPOINTER)&sFailOverCallbackContext, 0);
```

The contents of myFailOverCallback are as follows.

```
SQLUINTEGER myFailOverCallback(SQLHDBC aDBC,
void *aAppContext,
SQLUINTEGER aFailOverEvent)
{
 SQLHSTMT sStmt = SQL_NULL_HSTMT;
 SQLRETURN sRetCode;
 SQLINTEGER sVal;
 SQLLEN sLen;
 SQLUINTEGER sFailOverIntension = ALTIBASE_FO_GO;
 switch(aFailOverEvent)
 {
 case ALTIBASE_FO_BEGIN: // Fail-Over starts.
 break;
 case ALTIBASE_FO_END:
 sRetCode = SQLAllocStmt( aDBC,&sStmt);
 if(sRetCode != SQ_SUCCESS)
 {
 printf("FailOver-Callback SQLAllocStmt Error ");
 return ALTIBASE_FO_QUIT;
 }
 sRetCode = SQLBindCol(sStmt, 1, SQL_C_SLONG , &sVal,0,&sLen);
 if(sRetCode != SQ_SUCCESS)
 {
 printf("FailOver-Callback SQLBindCol");
 return ALTIBASE_FO_QUIT;
 }
 sRetCode = SQLExecDirect(sStmt, (SQLCHAR *) "SELECT 1 FROM DUAL",
 SQL_NTS);
 if(sRetCode != SQ_SUCCESS)
 {
 printf("FailOVer-Callback SQLExecDirect");
 return ALTIBASE_FO_QUIT;
```

```
}
while ( (sRetCode = SQLFetch(sStmt)) != SQL_NO_DATA )
{
if(sRetCode != SQL_SUCCESS)
{
printf("FailOver-Callback SQLBindCol");
sFailOverIntension = ALTIBASE_FO_QUIT;
break;
}
printf("FailOverCallback->Fetch Value = %d \n",sVal );
fflush(stdout);
}
sRetCode = SQLFreeStmt( sStmt, SQL_DROP );
ATC_TEST(sRetCode,"SQLFreeStmt");
break;
default:
break;
}//switch
return sFailOverIntension;
}//myFailOverCallback
```

4.4.3 Checking Whether Fail-Over Succeeded

If the result of SQLPrepare, SQLExecute, SQLFetch or the like is an error, rather than SQL_SUCCESS, a statement handle is handed over to SQLGetDiagRec, and if aNativeError has a diagnostic record equal to ALTIBASE_FAILOVER_SUCCESS, STF (Service Time Fail-Over) can be determined to have succeeded.

The following example demonstrates how to check whether STF (Service Time Fail-Over) was successful.

```
UInt isFailOverErrorEvent(SQLHSTMT aStmt)
{
 SQLRETURN rc;
 SQLSMALLINT sRecordNo;
 SQLCHAR sSQLSTATE[6];
 SQLCHAR sMessage[2048];
 SQLSMALLINT sMessageLength;
 SQLINTEGER sNativeError;
 UInt sRet = 0;
 sRecordNo = 1;
 while ((rc = SQLGetDiagRec(SQL_HANDLE_STMT,
 aStmt,
 sRecordNo,
 sSQLSTATE,
 &sNativeError,
 sMessage,
 sizeof(sMessage),
 &sMessageLength)) != SQL_NO_DATA)
 {
 sRecordNo++;
 if(sNativeError == ALTIBASE_FAILOVER_SUCCESS)
 {
 sRet = 1;
 break;
 }
 }
 return sRet;
}
```

The following example shows that when a network error occurs while SQLExecDirect is being executed, whether STF (Service Time FailOver) was successful is checked, and it is re-executed if necessary (in a prepare/execute environment, re-execution would have to start at the prepare stage).

```
retry:
 sRetCode = SQLExecDirect(sStmt,
(SQLCHAR *) "SELECT C1 FROM T2 WHERE C2 > ? ORDER BY C1",
SQL_NTS);
 if(sRetCode != SQL_SUCCESS)
 {
 if(isFailOverErrorEvent(sStmt) == 1)
 {
 goto retry;
 }
 else
 {
 printf("Error While DirectExeute....");
 exit(-1).
 }
 }
```

4.4.4 Example

4.4.4.1 Making Environment Settings

To implement the example, a data source called Test1 is described in altibase_cli.ini as follows.

```
[ Test1 ]
Server=192.168.3.53
Port=20300
User=altibase
Password= altibase
DataBase = mydb
AlternateServers=(192.168.3.54:20300,192.168.3.53:20300)
ConnectionRetryCount=3
ConnectionRetryDelay=5
LoadBalance = on
SessionFailOver = on
```

Additionally, the FailOverCallback function uses myFailOverCallback, which was described above.

When STF (Service Time Fail-Over) takes place, if it is successful, execution must be repeated starting with SQLPrepare (in the case of SQLDirectExecute, the prepare process is not necessary, and only SQLDirectExecute need be re-executed).

If STF (Service Time Fail-Over) occurs while data are being fetched, it will be necessary to call SQLCloseCursor and start over again from the prepare process (in the case of SQLDirectExecute, the prepare process is not necessary, and only SQLDirectExecute will need to be re-executed).

4.4.4.2 Sample Code

To view the complete contents of this example, please refer to $ALTIBASE_HOME/sample/SQLCLI/Fail-Over/FailOverCallbackSample.cpp, which should have been installed as part of the ALTIBASE HDB package.

```
#define ATC_TEST(rc, msg) if( ((rc)&(~1))!=0) { printf(msg); exit(1); }
//determining whether STF(Service Time FailOver) was successful.
UInt isFailOverErrorEvent(SQLHDBC aDBC,SQLHSTMT aStmt)
```

```
{
 SQLRETURN rc;
 SQLSMALLINTsRecordNo;
 SQLCHAR sSQLSTATE[6];
 SQLCHAR sMessage[2048];
 SQLSMALLINTsMessageLength;
 SQLINTEGERsNativeError;
 UInt sRet = 0;
 sRecordNo = 1;
 while ((rc = SQLGetDiagRec(SQL_HANDLE_STMT, aStmt,
 sRecordNo, sSQLSTATE,
 &sNativeError, sMessage,
 sizeof(sMessage),
 &sMessageLength)) != SQL_NO_DATA)
 {
 sRecordNo++;
 if(sNativeError == ALTIBASE_FAILOVER_SUCCESS)
 {
 sRet = 1;
 break;
 }
 }
 return sRet;
}
int main( SInt argc, SChar *argv[])
{
 SCharsConnStr[BUFF_SIZE] = {0};
 SQLHANDLE sEnv = SQL_NULL_HENV;
 SQLHANDLE sDbc = SQL_NULL_HDBC;
 SQLHSTMT sStmt = SQL_NULL_HSTMT;
 SQLINTEGER sC2;
 SQLRETURN sRetCode;
 SQLINTEGER sInd;
 SQLINTEGER sValue;
 SQLLEN sLen;
 UInt sDidCreate = 0;
 SChar sBuff[BUFF_SIZE2];
 SChar sQuery[BUFF_SIZE];
 SQLFailOverCallbackContext sFailOverCallbackContext;
 snprintf(sConnStr, sizeof(sConnStr), "DSN=Test1");
 sprintf(sQuery,"SELECT C1 FROM T2 WHERE C2 > ? ORDER BY C1");
 sRetCode = SQLAllocHandle(SQL_HANDLE_ENV, NULL, &sEnv);
 ATC_TEST(sRetCode,"ENV");
 sRetCode = SQLAllocHandle(SQL_HANDLE_DBC, sEnv, &sDbc);
 ATC_TEST(sRetCode,"DBC");
 /* connect to server */
 sRetCode = SQLDriverConnect(sDbc, NULL, (SQLCHAR *)sConnStr,
 SQL_NTS, NULL, 0, NULL,
 SQL_DRIVER_NOPROMPT);
 ATC_TEST(sRetCode,"SQLDriverConnect");
 sRetCode = SQLAllocStmt( sDbc,&sStmt);
 ATC_TEST(sRetCode,"SQLAllocStmt");
 sRetCode = SQLBindCol(sStmt, 1, SQL_C_CHAR , sBuff,BUFF_SIZE2,&sLen);
 ATC_TEST(sRetCode,"SQLBindCol");
 sRetCode = SQLBindParameter(sStmt, 1, SQL_PARAM_INPUT,
 SQL_C_SLONG, SQL_INTEGER,
 0, 0, &sC2, 0, NULL);
 ATC_TEST(sRetCode,"SQLBindParameter");
 sFailOverCallbackContext.mDBC = NULL;
 sFailOverCallbackContext.mAppContext = &sFailOverDirection;
 sFailOverCallbackContext.mFailOverCallbackFunc = myFailOverCallback;
 sRetCode = SQLSetConnectAttr(sDbc,ALTIBASE_FAILOVER_CALLBACK,
 (SQLPOINTER)&sFailOverCallbackContext,0);
 ATC_TEST(sRetCode,"SQLSetConnectAttr");
 retry:
```

```
sRetCode = SQLPrepare(sStmt, (SQLCHAR *)sQuery, SQL_NTS);
if(sRetCode != SQL_SUCCESS)
{
// If STF was successful, start over again from the prepare stage.
if(isFailOverErrorEvent(sDbc,sStmt) == 1)
{
goto retry;
}
else
{
ATC_TEST(sRetCode,"SQLPrepare");
}
}
sC2 = 0;
sRetCode = SQLExecute(sStmt);
if(sRetCode != SQL_SUCCESS)
{
// If STF was successful, start over again from the prepare stage.
if(isFailOverErrorEvent(sDbc,sStmt) == 1)
{
goto retry;
}
else
{
ATC_TEST(sRetCode,"SQLExecDirect");
}
}
while ( (sRetCode = SQLFetch(sStmt)) != SQL_NO_DATA )
{
if(sRetCode != SQL_SUCCESS)
{
if(isFailOverErrorEvent(sDbc,sStmt) == 1)
{
// If STF occurs during a fetch operation, it is absolutely essential to call
SQLCloseCursor.
SQLCloseCursor(sStmt);
goto retry;
}
else
{
ATC_TEST(sRetCode,"SQLExecDirect");
}
}
printf("Fetch Value = %s \n", sBuff);
fflush(stdout);
}
sRetCode = SQLFreeStmt( sStmt, SQL_DROP );
ATC_TEST(sRetCode,"SQLFreeStmt");
sRetCode = SQLDisconnect(sDbc);
ATC_TEST(sRetCode,"Disconnect()");
sRetCode = SQLFreeHandle(SQL_HANDLE_DBC, sDbc);
ATC_TEST(sRetCode,"Free HDBC");
sRetCode = SQLFreeHandle(SQL_HANDLE_ENV, sEnv);
ATC_TEST(sRetCode,"Free HENV");
}
```

4.5 WinODBC

4.5.1 Making Settings in .odbcinst.ini

In the DataSource section of .odbcinst.ini, the Threading property must be set to 1, and thread safety must be ensured at the statement level. Then, queries to check whether databases match can be executed by FailOverCallback during Fail-Over.

The following shows how to set the Threading property to 1 in the DataSource section for a data source called Test1:

```
[Test1]
Threading=1
```

4.5.2 Data Structures

When writing ODBC programs, the Fail-Over-related data structures used with CLI can be used without change; it is necessary only to include the sqlcli.h header which is provided with the client package of ALTIBASE HDB.

As can be seen below, the process of Fail-Over registration involves the creation of a FailOverCallbackContext object, and after DB Connect is successful, FailOverCallbackContext is populated with values.

Unlike when using CLI, in ODBC, SQLHDBC must be assigned from mDBC of the FailOverCallbackContext object.

4.5.3 Example

```
SQLFailOverCallbackContext sFailOverCallbackContext;
... -> some code is omitted here
 /* connect to server */
 sRetCode = SQLDriverConnect(sDbc, NULL,
 (SQLCHAR*)"DSN=127.0.0.1;UID=altibase;PWD=altibase;PORT_NO=20300;
 AlternateServers=(192.168.3.54:20300,192.168.3.53:20300);
ConnectionRetryCount=3;ConnectionRetryDelay=5;LoadBalance=on;
SessionFailOver=on;"),
 SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
 sFailOverCallbackContext.mDBC = sDbc;
 sFailOverCallbackContext.mAppContext = NULL;
 sFailOverCallbackContext.mFailOverCallbackFunc = myFailOverCallback;
 sRetCode = SQLSetConnectAttr(sDbc,ALTIBASE_FAILOVER_CALLBACK,
 (SQLPOINTER)&sFailOverCallbackContext,0);
```

4.6 Embedded SQL

Because the Fail-Over data structures used here are the same as those used in CLI, and because the structure of an ESQLC (Embedded SQL in C) application is similar to that of a CLI application, only the features unique to ESQLC will be described here.

4.6.1 Registering Fail-Over Callback Functions

Because SQLHDBC of CLI cannot be directly checked in an Embedded SQL program, the process of registering a Fail-Over callback function is as shown below.

Here, FailOverCallbackContext is declared in the declaration section.

```
EXEC SQL BEGIN DECLARE SECTION;
SQLFailOverCallbackContext sFailOverCallbackContext;
EXEC SQL END DECLARE SECTION;
```

FailOverCallbackContext is populated with values.

```
sFailOverCallbackContext.mDBC = NULL;
sFailOverCallbackContext.mAppContext = NULL;
sFailOverCallbackContext.mFailOverCallbackFunc = myFailOverCallback;
```

myFailOverCallback is the function that was seen in the CLI Fail-Over example above, only the CLI function and Os function need to be written, and Embedded SQL commands cannot be used.

The following shows how a Fail-Over Callback function is registered in an Embedded SQL statement.

```
EXEC SQL [AT CONNECTUON-NAME] REGISTER FAIL_OVER_CALLBACK :sFailOverCallback-
Context;
```

4.6.2 Checking Whether Fail-Over Succeeded

After the EXEC SQL command is executed, if the result of sqlca.sqlcode is ALTIBASE_FAILOVER_SUCCESS, rather than SQL_SUCCESS, then STF (Service Time Fail-Over) can be determined to have succeeded.

The following example demonstrates how to check whether STF (Service Time Fail-Over) was successful.

```
re-execute:
 EXEC SQL INSERT INTO T1 VALUES( 1 );
 if (sqlca.sqlcode != SQL_SUCCESS)
 {
 if (sqlca.sqlcode == ALTIBASE_FAILOVER_SUCCESS)
 {
 goto re-execute;
 }//if
 else
 {
 printf("SQLCODE : %d\n", SQLCODE);
 printf("sqlca.sqlerrm.sqlerrmc : %s\n", sqlca.sqlerrm.sqlerrmc);
 printf("%d rows inserted\n", sqlca.sqlerrd[2]);
 printf("%d times insert success\n\n", sqlca.sqlerrd[3]);
 }//else
 }
```

4.6.3 Example 1

```
main()
{
 EXEC SQL BEGIN DECLARE SECTION;
 SQLFailOverCallbackContext sFailOverCallbackContext;
 char sUser[10];
 char sPwd[10];
 char sConnOpt[1024];
 EXEC SQL END DECLARE SECTION;
 strcpy(sUser, "SYS");
 strcpy(sPwd, "MANAGER");
 sprintf(sConnOpt,"DSN=127.0.0.1;UID=altibase;PWD= altibase;PORT_NO=20300;
 AlternateServers=(192.168.3.54:20300,192.168.3.53:20300);ConnectionRetry-
Count=3;
 ConnectionRetryDelay=5;LoadBalance=on;SessionFailOver=on;"" );
 EXEC SQL CONNECT :sUser IDENTIFIED BY :sPwd USING : sConnOpt;
 if (sqlca.sqlcode != SQL_SUCCESS)
 {
 printf("SQLCODE : %d\n", SQLCODE);
 printf("sqlca.sqlerrm.sqlerrmc : %s\n", sqlca.sqlerrm.sqlerrmc);
 return 0;
 }
 else
 {
 printf("CONNECTION SUCCESS\n");
 }
 //FailOverCallbackContext is populated with values.
 sFailOverCallbackContext.mDBC = NULL;
 sFailOverCallbackContext.mAppContext = NULL;
 sFailOverCallbackContext.mFailOverCallbackFunc = myFailOverCallback;
 // FailOverCallbackContext is registered.
 EXEC SQL REGISTER FAIL_OVER_CALLBACK :sFailOverCallbackContext;
re-execute:
 EXEC SQL INSERT INTO T1 VALUES( 1 );
 if (sqlca.sqlcode != SQL_SUCCESS)
 {
 if (SQLCODE == EMBEDED_ALTIBASE_FAILOVER_SUCCESS)
 {
 goto re-execute;
 }//if
 else
 {
 printf("SQLCODE : %d\n", SQLCODE);
 printf("sqlca.sqlerrm.sqlerrmc : %s\n", sqlca.sqlerrm.sqlerrmc);
 printf("%d rows inserted\n", sqlca.sqlerrd[2]);
 printf("%d times insert success\n\n", sqlca.sqlerrd[3]);
 return 0;
 }//else
 }
 EXEC SQL DISCONNECT;
}
```

4.6.4 Example 2

This example demonstrates the use of a cursor. If Fail-Over occurs while a cursor is being used, EXEC SQL CLOSE RELEASE Cursor is executed, and the EXEC SQL DECLARE CURSOR statement is executed again, so that a new prepare process can be executed on an available server.

```
retry:
 EXEC SQL DECLARE CUR1 CURSOR FOR SELECT C1 FROM T2 ORDER BY C1;
```

```
if (sqlca.sqlcode == SQL_SUCCESS)
{
printf("DECLARE CURSOR SUCCESS.!!! \n");
}
else
{
if( SQLCODE == EMBEDED_ALTIBASE_FAILOVER_SUCCESS)
{
printf("Fail-Over SUCCESS !!! \n");
goto retry;
}
else
{
printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
return(-1);
}
}
EXEC SQL OPEN CUR1;
if (sqlca.sqlcode == SQL_SUCCESS)
{
printf("DECLARE CURSOR SUCCESS !!!\n");
}
else
{
if( SQLCODE == EMBEDED_ALTIBASE_FAILOVER_SUCCESS)
{
printf("Fail-Over SUCCESS !!! \n");
/* If a cursor is OPEN when Fail-Over occurs, the cursor must be closed
and released. */
EXEC SQL CLOSE RELEASE CUR1;
goto retry;
}
else
{
printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
return(-1);
}
} //else
while(1)
{
EXEC SQL FETCH CUR1 INTO :sC1;
if (sqlca.sqlcode == SQL_SUCCESS)
{
printf("Fetch Value = %s \n",sC1);
}
else if (sqlca.sqlcode == SQL_NO_DATA)
{
break;
}
else
{
if(SQLCODE == EMBEDED_ALTIBASE_FAILOVER_SUCCESS)
{
printf("DECLARE CURSOR SUCCESS !!!");
/* If a fetch operation is underway when Fail-Over occurs, the
cursor must be closed and released. */
EXEC SQL CLOSE RELEASE CUR1;
goto retry;
}
else
{
printf("Error : [%d] %s\n\n", SQLCODE, sqlca.sqlerrm.sqlerrmc);
return(-1);
}//else
}//else
```

```
}//while
EXEC SQL CLOSE CUR1;
```

Appendix A. FAQ

Replication FAQ

I want to know how to resolve conflicts.

Please refer to 2.3 Confilict Resolution.

Is replication possible between two servers located on different local networks?

Yes, it's possible. However, because of the great physical distance, replication performance may decrease somewhat in accordance with bandwidth and latency.

Can I execute ADD COLUMN on a replication target table?

Yes, you may execute DDL statements on replication target tables.

First, make the following property settings: set the REPLICATION_DDL_ENABLE property to 1, and, using the ALTER SESSION SET REPLICATION command, set the REPLICATION property to some value other than NONE.

For more information, please refer to 3.5 Executing DDL Statements on Replication Target Tables.

When one of two servers connected for replication goes down or offline and then comes back online, how can I check the current status of replication data to be sent to the other server?

The replication gap, meaning the number of redo logs for which corresponding XLogs need to be sent but have not yet been sent, can be checked by querying the REP_GAP column in the V$REPGAP performance view. Performance views can also be used to check various other information related to replication execution.

Is replication possible between two different kinds of servers?

Yes, it's possible. Heterogeneous replication function of ALTIBASE HDB takes into account byte ordering, structure aligning, endian and bit count on both the Sender and Receiver in order to make replication between different kinds of servers possible.

To achieve this, when XLOGs are sent or received, the Sender thread adds data to be sent to a transmission buffer, and the Receiver thread receives data from a reception buffer in the same order in which it was sent by the Sender thread. However, when performing replication between heterogeneous servers, if the byte order is different, the necessary operation of changing the byte order will entail a reduction in performance.

Can I add or delete tables while replication is active?

This is impossible while replication is underway. To add or delete replication target tables, it is first necessary to stop replication.

Can I perform replication between memory and disk tables?

Yes, it's possible.

Index