

ALTIBASE Administration

Replication Users' Manual

release 5.3.3



ALTIBASE Administration Replication User's Manual

Release 5.3.3

Copyright © 2001~2009 Altibase Corporation. All rights reserved.

This manual contains proprietary information of Altibase Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright patent and other intellectual property law. Reverse engineering of the software is prohibited.

All trademarks, registered or otherwise, are the property of their respective owners

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	i
About This Manual	ii
Target Users.....	ii
Software environment.....	ii
Organization.....	ii
Documentation Conventions	ii
Related Documents	v
Online Manual	v
Altibase Welcomes Your Opinions	v
1. Replication Overview.....	1
Introduction	2
Concepts.....	2
Terminology	2
How to Perform Replication in Altibase	3
Choosing a Replication Server.....	5
Choosing Replication Targets.....	5
Replication Mode	5
Replication of Partitioned Tables.....	5
Data Recovery Using Replication.....	6
2. Managing Replication	7
Replication Procedures.....	8
Troubleshooting	9
Abnormal shutdown of the local server.....	9
Interruption of communication between the local and remote servers.....	10
Network Failure.....	11
Conflict Resolution.....	12
User-Oriented Scheme	12
Master-Slave Scheme.....	13
Timestamp-Based Scheme.....	15
3. Deploying Replication	17
Considerations.....	18
Prerequisites.....	18
Data Requirements	18
Connection Requirements	18
Replication Target Column Constraints	18
Partitioned Table Constraints	19
Restrictions on Using Replication for Data Recovery	19
Additional Considerations when Using Replication for Data Recovery	19
Conditional Clause Requirements.....	19
Allowable DDL Statements	20
CREATE REPLICATION.....	21
Syntax	21
Description	21
Error Codes	22
Example	22
Starting, Stopping and Modifying Replication using "ALTER REPLICATION"	23
Syntax	23
Description	23
Error Codes	24
Example	25
DROP REPLICATION	26
Syntax	26
Description	26
Error Codes	26
Example	26

Executing DDL Statements on Replication Target Tables	27
Syntax	27
Description	27
Restrictions	27
Example	28
Extra Features.....	29
Recovery Option	29
Offline Option	30
Replication Conditional Clause	32
Description	32
Error codes	32
Examples.....	32
Restrictions	33
Replication in a Multiple IP Network Environment	34
Syntax	34
Description	34
Examples.....	35
Properties	39
4. Fail-Over	41
An Overview of Fail-Over	42
The Fail-Over Concept	42
The Fail-Over Process	43
Using Fail-Over	45
Registering Fail-Over Connection Properties.....	45
Checking Whether Fail-Over Has Succeeded.....	46
Writing Fail-Over Callback Functions.....	46
Writing Callback Functions for Use with JDBC.....	48
The JDBC Fail-Over Callback Interface	48
Writing Fail-Over Callback Functions for Use with JDBC	48
Checking Whether Fail-Over Has Succeeded in JDBC	50
Sending Fail-Over Connection Settings to WAS	50
JDBC Example	50
SQL CLI.....	53
SQL CLI-Related Data Structures.....	53
Registering Fail-Over in SQL CLI.....	54
Checking Whether Fail-Over Has Succeeded in SQL CLI.....	55
SQL CLI Example	56
WinODBC	59
WinODBC Data Structures	59
WinODBC Example.....	59
Embedded SQL.....	60
Registering Fail-Over Callback Functions in an Embedded Environment.....	60
Checking Fail-Over Success in an Embedded Environment.....	60
Embedded SQL Example	61
AppendixA. FAQ.....	65
Replication FAQ.....	65

Preface

About This Manual

This manual gives an overview of replication in Altibase and explains in detail how to perform replication.

Target Users

This manual has been prepared for the following Altibase 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 5.3.3 will be used as the database server.

Organization

This manual has been organized as follows:

- Chapter 1. Replication Overview
This chapter introduces replication in Altibase.
- Chapter 2. Managing Replication
This chapter explains replication procedures in Altibase.
- Chapter 3. Deploying Replication
This chapter explains how to establish a replication environment in Altibase.

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.






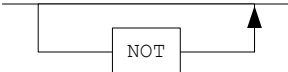
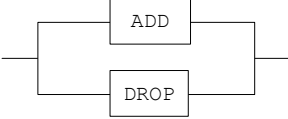
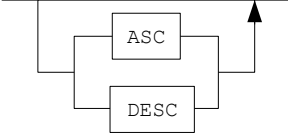
There are two sets of conventions:

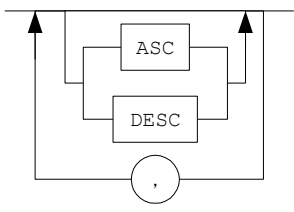
- syntax diagrams

- 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.
	Indicates an optional element comprised of options.

Elements	semantics
	<p>Indicates an optional element in which multiple elements may be specified. A comma must precede all but the first option.</p>

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.

Convention	Meaning	Example
[]	Indicates an optional item.	<pre> VARCHAR [(size)] [[FIXED] VARIABLE] </pre>
{ }	Indicates a mandatory field for which one or more items must be selected.	<pre> { ENABLE DISABLE COMPILE } </pre>
	A delimiter between optional or mandatory arguments.	<pre> { ENABLE DISABLE COMPILE } [ENABLE DIS- ABLE COMPILE] </pre>
...	Indicates that the previous argument is repeated, or that sample code has been omitted.	<pre> SQL> SELECT ename FROM employee; ENAME ----- - SWNO HJNO HSCHOI . . . 20 rows selected. </pre>
Other symbols	Symbols other than those shown above are part of the actual code.	<pre> EXEC :p1 := 1; acc NUMBER(11,2); </pre>
Italics	Statement elements in italics indicate variables and special values specified by the user.	<pre> SELECT * FROM table_name; CONNECT userID/password; </pre>

Convention	Meaning	Example
Lower Case Characters	Indicate program elements set by the user, such as table names, column names, file names, etc.	<code>SELECT ename FROM employee;</code>
Upper Case Characters	Keywords and all elements provided by the system appear in upper case.	<code>DESC SYSTEM_.SYS_INDICES_;</code>

Related Documents

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

- Altibase Installation User's Manual
- Altibase Administrator's Manual
- Altibase Starting User's Manual
- Altibase SQL User's Manual
- Altibase iSQL User's Manual
- Altibase Error Message Reference

Online Manual

Online versions of our manuals (PDF and HTML) are available from the Altibase Technical Center (<http://atc.altibase.com/>).

Altibase Welcomes Your Opinions

Please feel free to send us your comments and suggestions regarding this manual. Your comments and suggestions are important to us, and may be used to improve future versions of the manual. Please send your feedback to support@altibase.com, making sure to include the following information:

- The name and version of the manual in use
- Your comments and suggestions regarding the manual
- Your full name, address, and phone number

In addition to suggestions, this address may also be used to report any errors or omissions discovered in the manual, which we will address promptly. If you need immediate assistance with technical issues, please contact the Altibase Customer Support Center.

We always appreciate your comments and suggestions.

About This Manual

1 Replication Overview

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 Replication [Concepts](#) and [Terminology](#)
- [How to Perform Replication in Altibase](#)
- [Choosing Replication Targets](#)
- [Replication Mode](#)
- [Replication of Partitioned Tables](#)
- [Data Recovery Using Replication](#)

Concepts

The basic idea behind replication in Altibase is the use of the log replay method. To support the replication feature of Altibase, 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 also provides the Audit utility for monitoring and managing the replication status.

Terminology

- Local Server:
A server currently providing service. This is the Active Server in an active-passive configuration.
- Remote Server:
A Server operating for the purpose of replication. This is the passive server in an active-passive configuration.
- Sender Thread:
A thread on the local server 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.
- Receiver Thread :
A thread on the remote server that receives changed data sent from the local server.
- XSN: (XLOG Sequence Number)

The final position in a log file from which logs were transmitted by the replication Sender thread to the Receiver thread. When replication resumes, this position will be the position from which transmission will recommence.

- XLOG:

A log that transforms physical logs into logical form for replication. The replication Sender thread on a local server transmits the contents of an XLOG to the replication Receiver thread on a remote server so that the local server and remote server contain the same data.

- Master Transaction

A transaction that takes place on a local server and involves the execution of one or more DML statements on a replication target table or tables.

- Replicated Transaction

A transaction that occurs in response to the transmission of an XLOG, which is created based on a log that is generated in response to a master transaction, to the remote server.

How to Perform Replication in Altibase

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.

Aside from the 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 changed 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.

Figure [1-1] illustrates various ways that replication is supported. In Altibase, in consideration of performance and flexibility, the best of these ways is to transform physical logs into a directly executable logical structure.

Figure 1-1 [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, 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).

Choosing a Replication Server

In order to conduct replication in Altibase, the database character set must be a superset of the national character set. The character set can be checked by viewing `V$NLS_PARAMETERS` in Performance View.

Choosing Replication Targets

Altibase 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 viewing `V$REPRECEIVER_COLUMN` in Performance View.

Replication Mode

Altibase supports LAZY mode for replication.

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 replicated transactions to its DB.

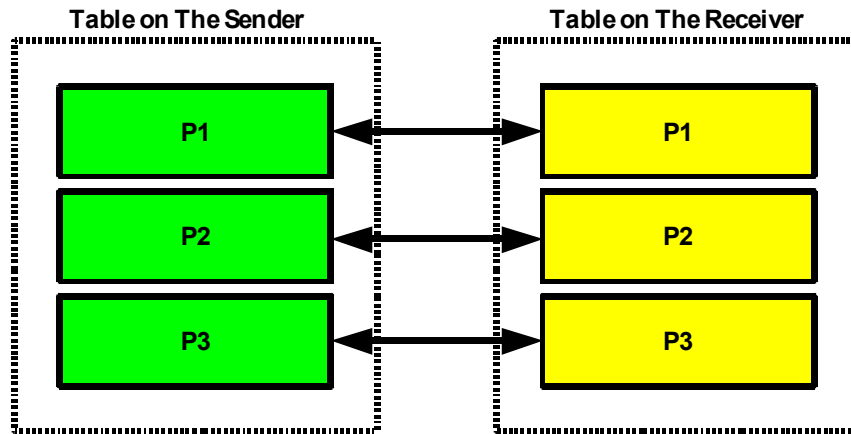
Therefore, because the service transaction and the replicated transaction take place in complete isolation from one another, the transactions do not influence one another, and the performance of the local server is excellent.

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

Replication of Partitioned Tables

As shown in [Figure 1-2], 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



Data Recovery Using Replication

Altibase 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 replicated transactions.

In Altibase, because data durability is somewhat compromised in the interests of performance, data are synchronized to ensure that no committed transactions disappear if a system shuts down abnormally.

2 Managing Replication

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

This chapter contains the following sections:

- [Replication Procedures](#)
- [Troubleshooting](#)
- [Conflict Resolution](#)

Replication Procedures

The following figure shows how replication works in Altibase.

Figure 2-1 Replication Procedures



1. Choose a server to replicate.
The database character set on this server must be a superset of the national character set.
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.
You must create a replication object that has 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.

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

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.
 4. The Receiver thread on Server B starts replication after being started by the Sender thread on Server A.

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 lowest XSN 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 having the lowest XSN.
 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.
- The Lowest XSN
 1. The lowest XSN is the lowest XLOG serial number corresponding to a transaction for which an XLOG for replication could not be sent.

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.

Conflict Resolution

To resolve data conflicts, Altibase supports three schemes for making data conflicts known to the user:

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

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.

Deferred Replication does not offer a perfect solution to data conflicts. At present, data conflicts are solved using a User-Oriented Scheme, a Master-Slave Scheme, a Timestamp-Based Scheme, or the like. The best way of completely resolving data conflicts when using Deferred Replication is to distribute the updated Data Set between systems.

In Altibase, the User-Oriented Scheme, the Master-Slave Scheme and the Timestamp-Based Scheme are used to resolve unavoidable data conflicts.

However, data conflicts affecting LOB columns cannot be resolved. This is because "Before Image" logging is not performed on LOB columns and primary and unique keys are not designated, 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.

User-Oriented Scheme

Syntax

```
CREATE REPLICATION replication_name
  WITH 'remote_host_ip', remote_host_port_no
  FROM user_name.table_name TO user_name.table_name [WHERE...],
  FROM user_name.table_name TO user_name.table_name [WHERE...],
  ...
  FROM user_name.table_name TO user_name.table_name [WHERE...],
```

Description

1. INSERT Conflict

If a transaction attempts to insert data having the same key as an existing record, it is not com-

mitted, and a conflict error message is recorded in `altibase_rp.log`.

2. DELETE Conflict

If a transaction attempts to delete data having a nonexistent key, it is not committed, and a conflict error message is recorded in `altibase_rp.log`.

3. UPDATE conflict

When an attempt is made to update a row having a value other than the expected value or having a nonexistent primary key, a conflict error message is output.

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. Depending on the application, the following policy can be used.

`REPLICATION_UPDATE_REPLACE=1` : Update

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

* For a detailed description of the CREATE REPLICATION command, please refer to the description of the [CREATE REPLICATION](#) command.

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.

Master-Slave Scheme

Syntax

```
CREATE REPLICATION replication_name {as master|as slave}
  WITH 'remote_host_ip', remote_host_port_no
  FROM user_name.table_name TO user_name.table_name [WHERE...],
  FROM user_name.table_name TO user_name.table_name [WHERE...],
  ...
  FROM user_name.table_name TO user_name.table_name [WHERE...];
```

Description

1. Specify "as master" or "as slave" in the command to specify whether the server is the Master or the Slave. If not specified, the value specified using the `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 handshake, the following combinations of `CONFLICT_RESOLUTION` field

Conflict Resolution

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 repl START
[ERR- : [REPL sender] failure to handshake with peer server (Replication
conflict resolution not allowed [1:0])]
```

* For a detailed information of the CREATE REPLICATION statement, please refer to the corresponding section.

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: The existing record is deleted and a new record is added.
 - UPDATE conflict: The conflict is ignored, and the transaction is committed regardless of the conflict.
 - DELETE conflict: Not committed.
 - Other: The XLOG transferred from the Master is processed as usual.

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, that there is a master-slave relationship between the local and remote servers, and that a table called “employee” and one called “department” are replication target tables. In this situation, replication is specified as follows:

- Local Server (IP: 192.168.1.60)

```
iSQL> CREATE REPLICATION repl AS MASTER
WITH '192.168.1.12',35524
FROM sys.employee TO sys.employee,
FROM sys.department TO sys.department;
Create success.
```
- Remote Server (IP: 192.168.1.12)

```
iSQL> CREATE REPLICATION repl AS SLAVE
WITH '192.168.1.60',25524
FROM sys.employee TO sys.employee,
FROM sys.department TO sys.department;
Create success.
```

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


```
iSQL> SELECT * FROM system.sys_replications;
SYS_REPLICATIONS_.REPLICATION_NAME SYS_REPLICATIONS_.LAST_USED_HOST_NO
-----
SYS_REPLICATIONS_.HOST_COUNT SYS_REPLICATIONS_.IS_STARTED
SYS_REPLICATIONS_.XSN_FILE_NO
-----
SYS_REPLICATIONS_.XSN_OFFSET SYS_REPLICATIONS_.ITEM_COUNT
SYS_REPLICATIONS_.CONFLICT_RESOLUTION
-----
REP1 3
1 0 -1
-1 2 1
1 row selected.
```

Timestamp-Based Scheme

Syntax

```
CREATE REPLICATION replication_name
  WITH 'remote_host_ip', remote_host_port_no
  FROM user_name.table_name TO user_name.table_name
  [WHERE...],
  FROM user_name.table_name TO user_name.table_name
  [WHERE...],
  ...
  FROM user_name.table_name TO user_name.table_name
  [WHERE...];
```

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:

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

Because AltiBase 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 '11.0,0,1', 30300 FROM sys.foo TO sys.foo, FROM
sys.bar TO sys.bar;
```

* For detailed information on the **CREATE REPLICATION** statement, please refer to the corresponding

section.

Timestamp-based Replication Processing Method

Altibase 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 (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.

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.

3 Deploying Replication

This chapter contains the following sections:

- [Considerations](#)
- [CREATE REPLICATION](#)
- [Starting, Stopping and Modifying Replication using "ALTER REPLICATION"](#)
- [DROP REPLICATION](#)
- [Executing DDL Statements on Replication Target Tables](#)
- [Extra Features](#)
- [Replication Conditional Clause](#)
- [Replication in a Multiple IP Network Environment](#)
- [Properties](#)

Considerations

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

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, except in the following cases:
 - deadlock: if a replication transaction is rolled back due to a state of deadlock on the remote server, an error message is not written to the log file.
 - network error: Data loss caused by a network (TCP/IP) error while replication is underway cannot be prevented. Additionally, no error message is written to the log 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.

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.
4. When SYNC is conducted on a memory table, there is no upper limit on the size of an XLOG. However, when SYNC is conducted on a disk table, an XLOG corresponding to one row must be smaller than 128 kB. (An XLOG used by SYNC comprises both a header, which is less than 1kB in size, and a row.)

Connection Requirements

1. There can be a maximum of 32 replication connections in one Altibase database.
2. The database character set and the national character set specified for replication should be the same as those set on the replication target database in order for replication to be possible. The character set that is currently in use can be checked by viewing NLS_CHARACTERSET, NLS_NCHAR_CHARACTERSET of V\$NLS_PARAMETERS.

Replication Target Column Constraints

1. When performing INSERT on a replication transaction, 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.

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.

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.

Additional Considerations when Using Replication for Data Recovery

1. If the data set to be updated is not perfectly divided between 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 property `REPLICATION_RECOVERY_MAX_TIME` made by the user, data might not be recovered.

Conditional Clause Requirements

The following considerations apply to the use of conditional clauses in replication:

1. The length of the conditional clause must be 1000 characters or less, including the "WHERE" keyword.
2. Operations that use comparison operators (<, >, <=, >=, ==, !=, <>) cannot be combined using logical operators in replication condition clauses.
3. In a relational operation, only one operand can be a column, and the other operand must be a constant.
4. Column operands must be data types that can be compared using comparison operators. BINARY, BLOB, CLOB, BIT, VARBIT, NIBBLE, GEOMETRY and the like can't be used as column operands.

Considerations

5. The name of the table to which the column operand belongs must be found in the FROM clause before the conditional clause.
6. Constant operands must be the same data type as the column operand.
7. The functions, subqueries, joins, and stored procedures supported in Altibase cannot be used as operands.

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
```

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.

CREATE REPLICATION

Before starting replication, information related to replication must first be correctly set.

Syntax

```
CREATE [LAZY|ACKED|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
[WHERE user_name.table_name.column_name {< | > | <> | >= | <= | = |
!=} value [{AND | OR} ... ]]
[,FROM user_name.table_name TO user_name.table_name [WHERE...]]
...;
```

Description

To perform replication, connection settings between the local and remote servers are made, replication is conducted on a table-by-table basis, and only one-to-one matching is possible. Connection can be made to a maximum of 32 different remote servers.

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

- *replication_name*
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*
Specifies whether the server is the Master or the Slave. If not specified, the value specified using the 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*
The IP address of the remote server.
- *remote_host_port_no*
The Port number at which the remote server Receiver thread listens. More specifically, the port number specified in REPLICATION_PORT_NO in the altibase.properties file.
- *user_name*
: The user ID with which to conduct replication.
- *table_name*

CREATE REPLICATION

: The name of the table to be replicated.

- *column_name*

The name of a column specified in a replication condition clause. This column must be found in the table specified in the FROM clause.

- *option_name*

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](#).

Error Codes

Please refer to the *Altibase Error Message Reference*.

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 “employee” and one called “department” between the two servers, the required replication definition would be as follows :

- Local server (IP: 192.168.60)

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

- Remote server (IP: 192.168.1.12)

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


Starting, Stopping and Modifying Replication using “ALTER REPLICATION”

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;
```

```
ALTER REPLICATION replication_name QUICKSTART;
```

```
ALTER REPLICATION replication_name STOP;
```

```
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
[WHERE user_name.table_name.column_name {< | > | <> | >= | <= | = |
!=} value [{AND | OR} ... ]];
```

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

```
ALTER REPLICATION replication_name SET MODE {LAZY|ACKED|EAGER};
```

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. However, when a table or tables are specified using the TABLE clause, because transactions involving that table(s) are processed after the completion of SYNC, changes to table(s) on which SYNC is individually executed will be suspended for some time before being processed.

- 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 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.)

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 current position in the log.
- 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 are deleted from all replication target tables, and then the SYNC is performed again.
- DROP TABLE

Because regular DDL statements cannot be executed on replication target tables, this function is provided so that replication target tables on which replication was stopped can be deleted.
- ADD TABLE

Because regular DDL statements cannot be executed on replication target tables, this function is provided so that replication target tables can be added to the database 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 changed log contents, up to the present log, to the other server. If used together with the ALL keyword, the current session waits so that changed log contents up until the most recent log, rather than the current log, can be sent to the other server.
- SET MODE

This changes the default mode that was specified at the time that the replication object was created defined, and can be executed when replication is stopped. One of LAZY, ACKED, or EAGER can be specified.

Error Codes

Please refer to the *Error Message Reference*.

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 transferring the data on the local server 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 bidirectional replication after it has started.

```
shell> server status replication
Admin> Connected with Altibase.
Admin> *-----*
* Replication *
*-----*
=== Sender List ===
  REPl[192.168.1.12,35524] (192.168.1.60,53475 => 192.168.1.12,35524)
Sender is running...
=== Sender List END ===
=== Replication Receiver List ===
  REPl(192.168.1.12,44990 => 192.168.1.60,25524) Receiver is running...
=== Replication Receiver List END ===
Admin> Good Bye!!
```

- 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.employee TO
sys.employee;
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.employee TO sys.employee;
Alter success.
```

DROP REPLICATION

Syntax

```
DROP REPLICATION replication_name;
```

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.

Error Codes

Please refer to the *Error Message Reference*.

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- : The replication has already been started.]
```

Executing DDL Statements on Replication Target Tables

Syntax

The DDL statements that Altibase 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 ...
```

Description

Altibase 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.

Restrictions

DDL statements cannot be executed on tables for which the replication recovery option has been specified.

The restrictions that govern the use of particular DDL statements are as follows.

- ALTER TABLE *table_name* ADD COLUMN cannot be used to add a column having a NOT NULL restriction .
 - A unique index cannot be added.
 - A foreign key cannot be added.
- ALTER TABLE *table_name* DROP COLUMN cannot be used to delete a column having a NOT NULL restriction.
 - A unique index cannot be added.
 - The primary key cannot be deleted.
 - A column used for filtering results in replication using a conditional clause cannot be deleted.
- CREATE INDEX
 - This is supported only for indexes that are not unique.

Executing DDL Statements on Replication Target Tables

- DROP INDEX

This is supported only for indexes that are not unique.

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.
```

Extra Features

Altibase provides the following extra replication features:

- Recovery Option
- [Offline Option](#)

Recovery Option

Syntax

```
ALTER REPLICATION replication_name SET RECOVERY ENABLE;
ALTER REPLICATION replication_name SET RECOVERY DISABLE;
```

Description

One of the extra replication features that Altibase 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.

Restriction

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

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.
```

Offline Option

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;
```

Description

One of the other extra replication features provided with Altibase 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 Active Server develops a fault, the Standby Server directly accesses the log directory on the Active Server using the Offline option, so that it can implement the logs that could not be sent.

If the `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.

Offline Option Restrictions

- 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.

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 'active_server/  
altibase_home/logs'  
WITH '127.0.0.1',33000 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;
```

Replication Conditional Clause

Replication conditions can be specified in a WHERE clause as long as those conditions pertain to a table defined in a FROM clause. The Sender thread transmits to the remote server only those XLOGs that satisfy the conditions specified in the condition clause.

Description

When performing replication, it is possible to replicate only a portion of the data if a conditional clause is specified.

For example, for a business with several branches, replication can be used to reconcile the data from each branch with the data from head office. The head office summarizes the data from all branches in one table for ease of management. In the case where information from a branch changes, all of its information must be sent to head office. However, when the range of information that has been changed or is not synchronized is known, a conditional clause can be created and used so that the data can be synchronized merely by sending only the changed data.

However, depending on whether the data in a column specified in a replication condition clause ("condition column") have changed, whether the conditions in the condition clause are satisfied can vary. Depending on this, the following actions would be conducted on the remote server:

- In the case where data in a condition column satisfy the condition before the change, but no longer satisfy the condition after the change:

On the remote server, the data in a condition column satisfy the condition. However, if the data in a condition column are changed and then do not satisfy the condition, the primary key data on the remote server are deleted.

- In the case where data in a condition column do not satisfy the condition before the change, but satisfy the condition after the change:

Because the data in the condition column satisfy the replication conditions, primary key data are inserted on the remote server.

Error codes

Please refer to the *Error Message Reference*.

Examples

- Creating a replication object based on a conditional clause:

```
iSQL> CREATE REPLICATION rep1 WITH '127.0.0.1', 47146
FROM sys.t1 TO sys.t1 WHERE id >= CHAR'a' AND id <= CHAR'z',
FROM sys.t2 TO sys.t2 WHERE join_date >= TO_DATE('2008-01-01', 'YYYY-MM-DD'),
FROM sys.t3 TO sys.t3 WHERE (age >= INTEGER'18' OR gender = CHAR'M')
AND address = VARCHAR'seoul';
```

- Adding a table and a condition clause to an existing replication object:

```
iSQL>ALTER REPLICATION repl ADD TABLE  
FROM sys.t1 TO sys.t1 WHERE id > CHAR'a' AND id <= CHAR'z';
```

Restrictions

- It is preferable to use a primary key column as a condition column. When using an ordinary column as a condition column, it is necessary to ensure that the data in the conditional column are not changed in order to prevent logging-related overhead.
- It is also necessary to ensure that the data in the conditional column are not changed when performing replication in Active-Active mode.

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 more than two physical network connections.

Syntax

- ```
CREATE REPLICATION replication_name {as master|as 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;
```

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.

Examples

In the following double-IP network environment, a replication object having a table called "employee" and another called "department" as its 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.employee TO sys.employee,
FROM sys.department TO sys.department;
Create success.<- Replication created in 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.employee TO sys.employee,
FROM sys.department TO sys.department;
Create success.<- Replication object 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.
SYS_REPLICATIONS_.REPLICATION_NAME SYS_REPLICATIONS_.LAST_USED_HOST_NO
-----
SYS_REPLICATIONS_.HOST_COUNT SYS_REPLICATIONS_.IS_STARTED
SYS_REPLICATIONS_.XLS
-----
SYS_REPLICATIONS_.ITEM_COUNT SYS_REPLICATIONS_.CONFLICT_RESOLUTION
-----
REP1 2
2 0 -1
2 0
1 row selected.
iSQL> SELECT * FROM system_.sys_repl_hosts_; <- The meta table enables the user to
view the remote server-related information.
SYS_REPL_HOSTS_.HOST_NO SYS_REPL_HOSTS_.REPLICATION_NAME
-----
SYS_REPL_HOSTS_.HOST_IP SYS_REPL_HOSTS_.PORT_NO
-----
2 REP1
192.168.1.154 30570
3 REP1
192.168.2.154 30570
2 rows selected.
iSQL> ALTER REPLICATION rep1 START;
Alter success.<- Replication starts
shell> server status replication; <- The status of replication is checked after replication
starts. (The Sender thread connects to the peer using the first IP and PORT.
Admin> Connected with Altibase.
Admin> *-----*
* Replication *
*-----*
```

Replication in a Multiple IP Network Environment

```
=== Sender List ===
REP1[192.168.1.154,30570] (192.168.1.51,25236 => 192.168.1.154,27255)
Sender is running...
=== Sender List END ===
=== Replication Receiver List ===
=== Replication Receiver List END ===
Admin> Good Bye!!
!!!!!!!!!!!!!! Network line disconnection !!!!!!!!!!!!!!!
shell> SERVER STATUS REPLICATION; <- The status of replication is checked after network failure occurs. This verifies reconnection using the second IP and PORT.
Admin> Connected with Altibase.
Admin> *-----*
* Replication *
*-----*
=== Sender List ===
REP1[192.168.2.154,30570] (192.168.1.51,29332 => 192.168.2.154,27255)
Sender is running...
=== Sender List END ===
=== Replication Receiver List ===
=== Replication Receiver List END ===
Admin> Good Bye!!
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.<- Replication is stopped
iSQL> ALTER REPLICATION rep1 START;
Alter success.<- Replication starts
shell> server status replication; <- When replication is started again after having been stopped, it can be verified to have been reconnected to the IP and PORT to which it was connected before being stopped.
Admin> Connected with Altibase.
Admin> *-----*
* Replication *
*-----*
=== Sender List ===
REP1[192.168.2.154,30570] (192.168.1.51,32916 => 192.168.2.154,27255)
Sender is running...
=== Sender List END ===
=== Replication Receiver List ===
=== Replication Receiver List END ===
Admin> Good Bye!!
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.
shell> server status replication; <- Connection to the peer using the newly designated IP 192.168.1.154 and PORT number 30570 can be confirmed.
Admin> Connected with Altibase.
Admin> *-----*
* Replication *
*-----*
=== Sender List ===
REP1[192.168.1.154,30570] (192.168.1.51,35732 => 192.168.1.154,27255)
Sender is running...
=== Sender List END ===
=== Replication Receiver List ===
=== Replication Receiver List END ===
Admin> Good Bye!!
```

- The following messages are written to altibase_rp.log during execution of the above-men-

tioned example.

- By setting the Trace log value to 1, it is possible to check whether the HeartBeat Thread was active (ALTER SYSTEM SET TRCLOG_SET_HBT_LOG = 1);
- The following message is written to the log file after a replication object is created and started. Whether the corresponding host has failed is checked at intervals corresponding to REPLICATION_HB_DETECT_TIME, which in this case has been set to 3 seconds.

```
[2003/07/03 16:38:50] == Network Fault Detection Proceeding ==
[2003/07/03 16:38:53] == Network Fault Detection Proceeding ==
[2003/07/03 16:38:56] == 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.

```
[2003/07/03 16:39:32] == Network Fault Detection Proceeding ==
[192.168.1.154:30570] Ref=1 Mode=1 Handle=2 WaterMark=0 mFault = No Fault
[2003/07/03 16:39:35] == Network Fault Detection Proceeding ==
[192.168.1.154:30570] Ref=1 Mode=0 Handle=2 WaterMark=0 mFault = No Fault
...
```

- If the REPLICATION_HBT_DETECT_HIGHWATER_MARK, which is one of the Altibase 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.

```
[2003/07/03 16:41:05] == Network Fault Detection Proceeding ==
[192.168.1.154:30570] Ref=1 Mode=1 Handle=2 WaterMark=1 mFault = No Fault
[2003/07/03 16:41:08] == Network Fault Detection Proceeding ==
[192.168.1.154:30570] Ref=1 Mode=1 Handle=2 WaterMark=2 mFault = No Fault
[2003/07/03 16:41:11] == Network Fault Detection Proceeding ==
[192.168.1.154:30570] Ref=1 Mode=1 Handle=2 WaterMark=3 mFault = No Fault
...
[2003/07/03 16:41:32] == Network Fault Detection Proceeding ==
[192.168.1.154:30570] Ref=1 Mode=1 Handle=2 WaterMark=10 mFault = No
Fault
[2003/07/03 16:41:35] == Network Fault Detection Proceeding ==
[192.168.1.154:30570] Ref=1 Mode=2 Handle=2 WaterMark=11 mFault = Yes
Fault!!
...
[2003/07/03 16:41:35] [Thr:98326] [REPL] getNextLastUsedHostNo:
192.168.1.154 : 30570 => 192.168.2.154 : 30570
[2003/07/03 16:41:38] == Network Fault Detection Proceeding ==
...
[192.168.2.154:30570] Ref=1 Mode=1 Handle=13 WaterMark=0 mFault = No
Fault
[2003/07/03 16:41:50] == Network Fault Detection Proceeding ==
[192.168.2.154:30570] Ref=1 Mode=0 Handle=13 WaterMark=0 mFault = No
Fault
[2003/07/03 16:41:53] == Network Fault Detection Proceeding ==
[192.168.2.154:30570] Ref=1 Mode=1 Handle=13 WaterMark=0 mFault = No
Fault
...
```

- The following message will be output when replication stops:

```
[2003/07/03 16:49:44] == Network Fault Detection Proceeding ==
[2003/07/03 16:49:47] == Network Fault Detection Proceeding ==
```

Replication in a Multiple IP Network Environment

...

Properties

To use replication, the Altibase properties file should be modified to suit the purposes of the user. The following properties are described in the *Altibase 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_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_PROPAGATION
- 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

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 provided with Altibase and how to use it.

- [An Overview of Fail-Over](#)
- [Using Fail-Over](#)
- [Writing Callback Functions for Use with JDBC](#)
- [SQL CLI](#)
- [WinODBC](#)
- [Embedded SQL](#)

An Overview of Fail-Over

The Fail-Over 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 became 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 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.

Altibase’s Fail-Over settings 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.

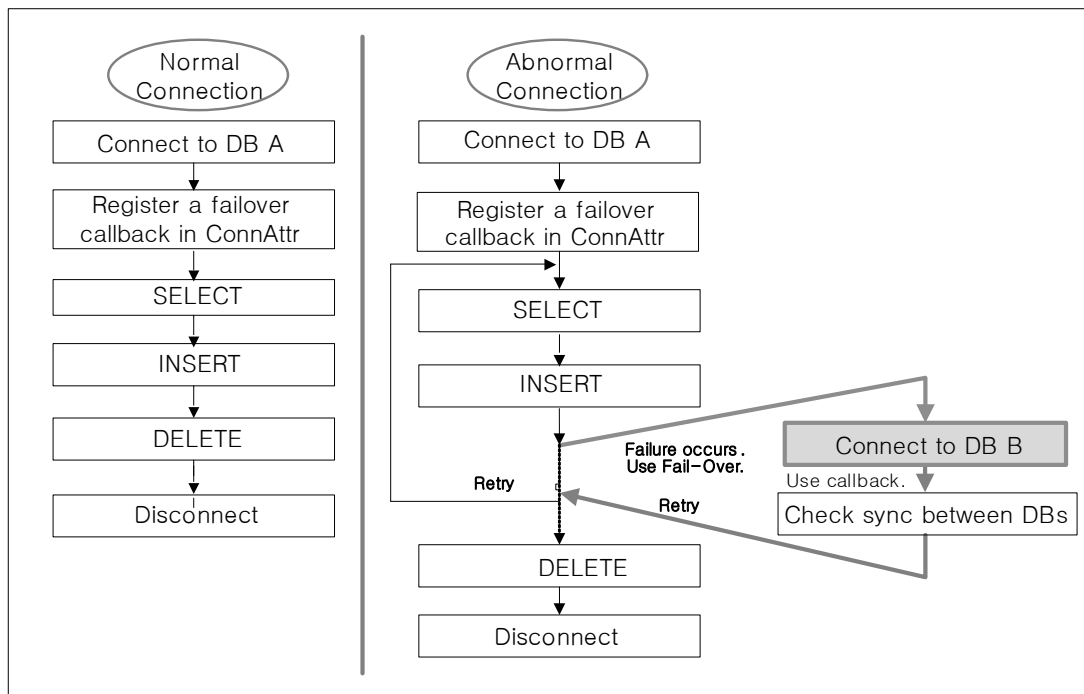
The three kinds of Fail-Over-related tasks that must be executed by the client application are summarized as follows:

- 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

The Fail-Over Process

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

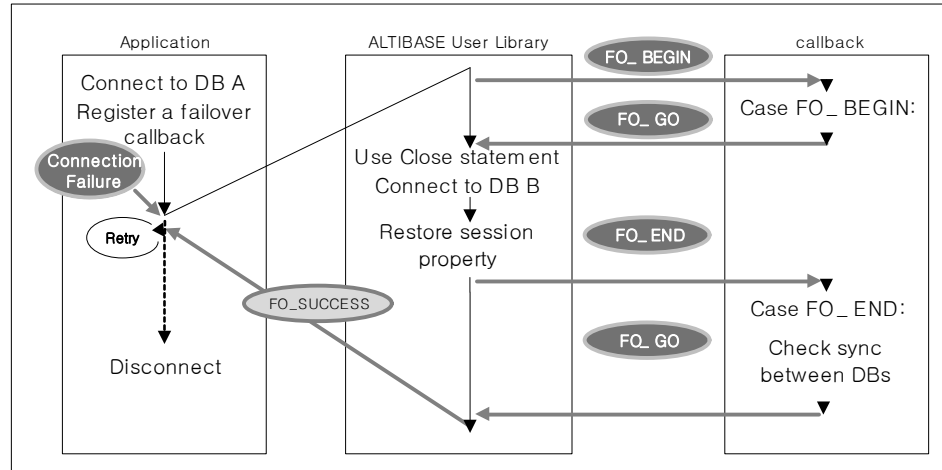
Figure 4-1 Fail-Over Registration and Handling Process



Fail-Over Callback must be registered by the user, and, once registered, during the Fail-Over process the Altibase 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 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 User Library connects and logs in to that database. Additionally, the properties of the previous session (auto-commit 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.

Using Fail-Over

Registering Fail-Over Connection Properties

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

The properties can be viewed in the following two ways:

- by checking the Connection String using the API's connection function
- by checking the appropriate Altibase settings file (altibase_cli.ini or odbci.ini [WinODBC])

Checking the Connection String

When the connection function is executed in the client application, the following is output:

[JDBC]

```
Jdbc:Altibase://192.168.3.51:20300/mydb?AlternateServers=(192.168.3.54:20300,192.168.3.53:20300) &
ConnectionRetryCount=3&ConnectionRetryDelay=3&LoadBalance=off&Session-
FailOver=on";
```

[ODBC, Embedded SQL]

```
DSN=192.168.3.51;UID=altibase;PWD=altibase;PORT_NO=20300;
AlternateServers=(192.168.3.54:20300,192.168.3.53:20300);
ConnectionRetryCount=3;ConnectionRetryDelay=5;LoadBalance=on;Session-
FailOver=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.

Displaying the Settings File

The Fail-Over connection settings are indicated 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.

Using Fail-Over

```
[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.

Checking Whether Fail-Over Has 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 `SQLGetDiagRec`, and if a `NativeError` 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.

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.

Writing Callback Functions for Use with JDBC

The JDBC 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.

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.

Writing Fail-Over Callback Functions for Use with JDBC

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 before Fail-Over continues and 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();

```

Writing Callback Functions for Use with JDBC

```
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);
```

Checking Whether Fail-Over Has Succeeded in JDBC

Checking whether Fail-Over, particularly STF (Service Time Fail-Over), was successful is conducted using `SQLException`. 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.

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;  
  }  
}
```

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";
```

JDBC 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 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`. A value of `SQLStates.ES_08FO01` 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.
        ((ABCConnection)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_08F001]) == true)
                {
                    continue;
                }
                System.out.println( "EXCEPTION : " + e.getMessage() );
                break;
            }
            break;
        } // while
        */

        while(true)
        {
            try

```

Writing Callback Functions for Use with JDBC

```
{
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_08F001]) == true)
{
continue;
}
System.out.println( "EXCEPTION : " + e.getMessage() );
break;
}
break;
}
sRes.close();
//-----
// Finalize
//-----

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

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.

SQL CLI-Related Data Structures

The prototype of the `FailOverCallback` 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 `FailOver Callback Context`, 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 `FailOverCallback`.

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

ALTIBASE_FO_BEGIN: 0

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

ALTIBASE_FO_END: 1

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

ALTIBASE_FO_ABORT: 2

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

ALTIBASE_FO_GO: 3

`FailOverCallback` sends `aFailOverEvent` to the CLI library so that STF can advance to the next step.

ALTIBASE_FO_QUIT: 4

`FailOverCallback` sends `aFailOverEvent` to the CLI library to prevent STF from advancing to the next step.

`FailOverCallbackContext` 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.

Registering Fail-Over in SQL CLI

As can be seen below, the process of Fail-Over registration involves the creation of FailOverCallbackContext, 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.

```
SQLUIINTEGER myFailOverCallback(SQLHDBC aDBC,
void *aAppContext,
SQLUIINTEGER aFailOverEvent)
{
SQLHSTMT sStmt = SQL_NULL_HSTMT;
SQLRETURN sRetCode;
SQLINTEGER sVal;
SQLLEN sLen;
SQLUIINTEGER 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

```

Checking Whether Fail-Over Has Succeeded in SQL CLI

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 exe-

SQL CLI

cuted, 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 DirectExecute....");
            exit(-1);
        }
    }
}
```

SQL CLI Example

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).

SQL CLI 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 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;
    sprintf(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:

```

SQL CLI

```
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");
}
```

WinODBC

WinODBC 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 Altibase client package.

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

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

An example follows:

```
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);
```

WinODBC Example

sqlcli.h must be included when writing an ODBC application, and when registering Fail-Over, as was described above in the section pertaining to CLI, SQLHDBC of ODBC must be assigned from mDBC FailOverCallbackContext.

Making Settings in .odbcinst.ini

In the DataSource section of .odbcinst.ini, Threading must be set to 1, thread safety must be assured at the statement level, and 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
```

Embedded SQL

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

Registering Fail-Over Callback Functions in an Embedded Environment

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 a 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 CONNECTION-NAME] REGISTER FAIL_OVER_CALLBACK :sFailOverCallback-
Context;
```

Checking Fail-Over Success in an Embedded Environment

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
}
```

Embedded SQL Example

Embedded SQL 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 == EMBEDDED_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;
}

```

Embedded SQL 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;

```

Embedded SQL

```
if (sqlca.sqlcode == SQL_SUCCESS)
{
printf("DECLARE CURSOR SUCCESS.!!! \n");
}
else
{
if( SQLCODE == EMBEDDED_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 == EMBEDDED_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 == EMBEDDED_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

Question

I want to know how to resolve conflicts.

Answer

When INSERT and DELETE conflicts occur, Altibase ignores them, but logs error messages. In the event of an UPDATE conflict, Altibase skips the XLOG if REPLICATION_UPDATE_REPLACE is set to 0, whereas the update operation continues without regard to the error if REPLICATION_UPDATE_REPLACE is set to 1.

Question

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

Answer

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

Question

Can I execute ADD COLUMN on a replication target table?

Answer

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 [Executing DDL Statements on Replication Target Tables](#).

Question

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?

Answer

The replication gap, meaning the data for replication that needs to be sent but has not yet been sent, can be checked by viewing the REP_GAP property in V\$REPGAP in Performance View. Performance Views can also be used to check various other information related to replication execution.

Question

Is replication possible between two different kinds of servers?

Answer

Yes, it's possible. Altibase's heterogeneous replication function 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.

Question

Can I add or delete tables during replication?

Answer

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

Question

Can I perform replication between memory and disk tables?

Answer

Yes, it's possible.

Index

A

ADD HOST 34
ADD TABLE Clause 24
ALTER REPLICATION 23

C

Checking Whether Fail-Over Has Succeeded 46
Communication Channel Error 10
Conflict Resolution 12
CREATE REPLICATION 21
CTF 42

D

Delete Conflict 13
DROP HOST 34
Drop replication 26
DROP TABLE Clause 24

F

Fail-Over Concept 42
Fail-Over Interface

- Embedded SQL 60
- JDBC 48
- SQLCLI 53
- WinODBC 59

Fail-Over Process 43
FAQ 65
FLUSH 24

I

Insert Conflict 12

L

local server 2

M

Master-slave Scheme 13

P

PARALLEL Clause 23

Q

QUICKSTART 24

R

receiving thread 2
Replication Definition 2

Replication in Multi-IP Network Environment 34
Replication property 39

S

Server Crash 9
SET HOST 35
START 24
STF 42
STOP 24
SYNC 23
SYNC ONLY 24

T

Timestamp-based Scheme 15
Transmission thread 2
Troubleshooting Replication Problems 9

U

Update Conflict 13
User-oriented Scheme 12
Using Fail-Over 45

X

XSN 2