Altibase Application Development

Log Analyzer User's Manual

Release 5.5.1

January 15, 2013

Altibase Application Development Log Analyzer User's Manual
Release 5.5.1

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

Contents

Preface

About This Manual

This manual describes the concept of the ALTIBASE® HDB™ Log Analyzer and explains how to use it.

Audience

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

- Database administrators
- Performance managers
- Database users
- Application developers
- Programmers
- Technical support workers

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

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

Software Environment

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

Organization

This manual is organized as follows:

- Chapter1: Introduction

  This chapter describes the concept behind the ALTIBASE HDB Log Analyzer and explains in simple terms how to use it.

- Chapter2: The XLog Sender

  This chapter explains how to use the XLog Sender, which is one of the components of the ALTIBASE HDB Log Analyzer.

- Chapter3: Analyzing XLogs

  This chapter describes XLogs, meta data and the ALTIBASE HDB internal data types, all of

which are required in order to analyze XLogs.

- Chapter4: Log Analysis API

  This chapter describes how to use the Log Analysis API component of the ALTIBASE HDB Log Analyzer.

- Appendix A. Error Codes

  This appendix describes the Log Analyzer error codes and the cause of each kind of error.

- Appendix B. Sample Code

  This appendix explains the location of the sample applications and provides a brief description of the use of the Log Analyzer with reference to sample code.

Documentation Conventions

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

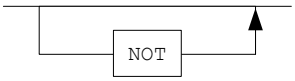
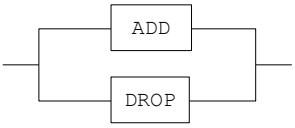
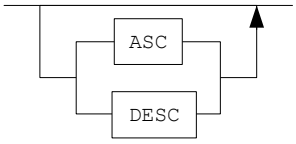
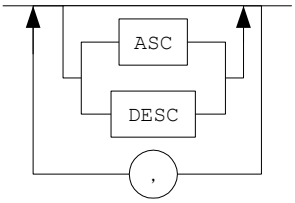
There are two sets of conventions:

- Syntax Diagram Conventions
- Sample Code Conventions

Syntax Diagram Conventions

In this manual, the syntax of commands is described using diagrams composed of the following elements:

Element	Description
	Indicates the start of a command. If a syntactic element starts with an arrow, it is not a complete command.
	Indicates that the command continues to the next line. If a syntactic element ends with this symbol, it is not a complete command.
	Indicates that the command continues from the previous line. If a syntactic element starts with this symbol, it is not a complete command.
	Indicates the end of a statement.

Element	Description
	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.
	Indicates an optional element in which multiple elements may be specified. A comma must precede all but the first element.

Sample Code Conventions

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

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

Convention	Meaning	Example
[]	Indicates an optional item.	`VARCHAR [(size)] [[FIXED |] VARIABLE]`

Convention	Meaning	Example
{}	Indicates a mandatory field for which one or more items must be selected.	{ ENABLE | DISABLE | COMPILE }
|	A delimiter between optional or mandatory arguments.	{ ENABLE | DISABLE | COMPILE } [ENABLE | DISABLE | COMPILE]
. . .	Indicates that the previous argument is repeated, or that sample code has been omitted.	iSQL> select e_lastname from employees; E_LASTNAME ------------------------ Moon Davenport Kobain . . . 20 rows selected.
Other symbols	Symbols other than those shown above are part of the actual code.	EXEC :p1 := 1; acc NUMBER(11,2);
Italics	Statement elements in italics indicate variables and special values specified by the user.	SELECT * FROM table_name; CONNECT *userID*/*password*;
Lower Case Letters	Indicate program elements set by the user, such as table names, column names, file names, etc.	SELECT e_lastname FROM employees;
Upper Case Letters	Keywords and all elements provided by the system appear in upper case.	DESC SYSTEM_.SYS_INDICES_;

Related Reading

For additional technical information, please refer to the following manuals:

- ALTIBASE HDB Getting Started Guide
- ALTIBASE HDB Administrator's Manual
- ALTIBASE HDB Replication Manual
- ALTIBASE HDB SQL Reference
- ALTIBASE HDB ODBC Reference
- ALTIBASE HDB Spatial SQL Reference
- ALTIBASE HDB Application Program Interface User's Manual
- ALTIBASE HDB iSQL User's Manual
- ALTIBASE HDB Error Message Reference

Online Manuals

Online versions of our manuals (PDF or HTML) are available from the Altibase Download Center (http://atc.altibase.com/).

Altibase Welcomes Your Comments

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.

When you send your feedback, please make sure to include the following information:

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

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

For immediate assistance with technical issues, please contact the Altibase Customer Support Center.

We always appreciate your comments and suggestions.

1 Introduction

This chapter describes the concept behind the ALTIBASE HDB Log Analyzer and explains in simple terms how to use it.

1.1 ALTIBASE HDB Log Analyzer

The ALTIBASE HDB Log Analyzer comprises a module that is included with ALTIBASE HDB for use in providing a history of database DML transactions on the basis of active logs, an external module that is connected to the internal module, and an API that is provided for using XLogs.

The ALTIBASE HDB Log Analyzer can be used:

1. to link ALTIBASE HDB with other DBMS products, and
2. to detect changes within ALTIBASE HDB from outside so as to respond to these changes in the desired manner.

1.1.1 Terms & Concepts

1.1.1.1 XLog

An XLog is a logical form of log that results from the conversion of a physical log. It is used to store a history of transactions involving DML (INSERT, UPDATE and DELETE) statements such that it can be accessed by users.

1.1.1.2 XLog Sender

The XLog Sender is the module that analyzes active logs to create XLogs, which it then passes to the XLog Collector.

The XLog Sender actively performs handshaking and XLog transmission.

1.1.1.3 XLog Collector

The XLog Collector is the module that receives meta data and XLogs from the XLog Sender.

The XLog Collector comprises meta data, an XLog queue, a transaction table and an XLog pool. It is called via the Log Analysis API.

1.1.1.4 Log Analysis API

The Log Analysis API is used to obtain XLogs and meta data that are used to interpret the XLogs.

1.1.1.5 Handshaking

Handshaking is the task of checking the protocol version, meta data, etc. before the XLogs are sent from the XLog Sender to the XLog Collector.

1.1.1.6 XLog Queue

The XLog Queue is the place where available XLogs are stored before they are accessed by users.

1.1.1.7 XLog Pool

The XLog Pool is memory that has been allocated for the storage of XLogs.

The purpose of the XLog Pool is to reuse the memory that has been allocated to XLogs in order to prevent excessive memory usage.

1.1.1.8 Transaction Table

The transaction table is the place where information about transactions, including the status of transactions, is stored.

1.1.1.9 Restart SN

The Restart SN is the SN of the active log from which reading will resume when the XLog Sender is restarted.

1.1.1.10 SN

The SN (Sequence Number) is the serial number of an individual log record.

1.1.1.11 Replication

Replication is a module that is used to synchronize data between two Altibase databases. For more information, please refer to the *Replication Manual*.

1.1.1.12 Replication SYNC

Replication SYNC is a replication function whereby all of the records in the replication target tables on the local server are sent to the remote server.

This feature is used to synchronize replication target tables on two databases before active log-based replication is started. For more information, please refer to the description of the ALTER REPLICATION statement in the *Replication Manual*.

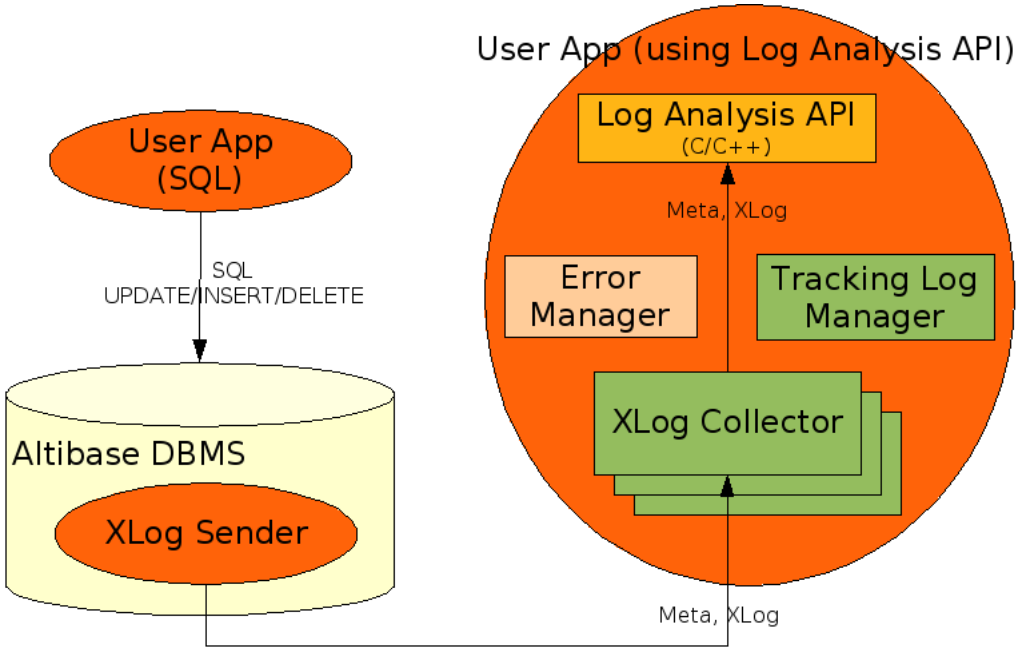
1.1.2 How the ALTIBASE HDB Log Analyzer Works

The location of the XLog Sender is within an Altibase database. It creates XLogs based on active logs and sends the XLogs along with related meta data to the XLog Collector. The XLog Collector exists within a client application, and provides end users with the XLogs and meta data via the Log Analysis API.

If for some reason a call to the Log Analysis API fails, it is of course necessary to take suitable measures, depending on the cause of the error. The most recent error information is stored in the Error Manager. Additionally, the Log Manager is provided for use in tracing errors. The Log Manager records simple trace and error information in the log files that are specified when the Log Manager is created.

The overall structure is shown in the following illustration:

Figure 1-1 The Structure of the ALTIBASE HDB Log Analyzer



The Log Analysis API can be used to obtain XLogs and related meta data from the XLog Collector. The XLog Collector receives these meta data from the XLog Sender at the time of handshaking between the XLog Sender and the XLog Collector. The meta data are valid until the next time handshaking is performed.

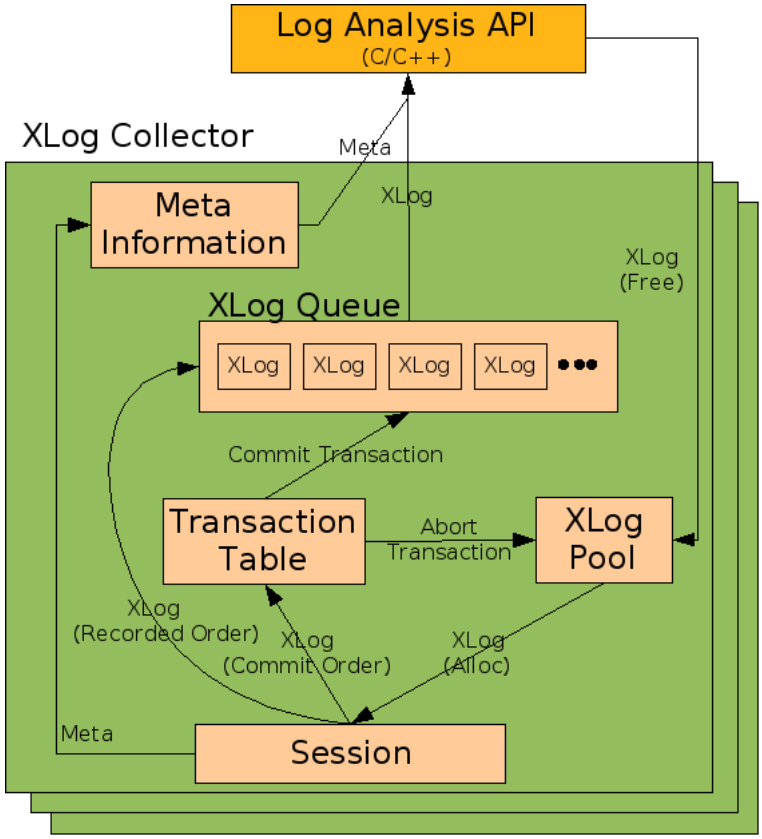
The movement of XLogs and meta data within the XLog Collector is as follows:

1. Memory for XLogs is obtained from the XLog Pool.
2. The XLog Collector receives the data that constitute XLogs from the XLog Sender and uses the data to create XLogs.
3. The XLogs are added to the XLog Queue, where they are accessed and used via the Log Analysis API.

   When it is desired to obtain the XLogs for transactions in the order in which the transactions are committed, the XLogs are temporarily stored in the Transaction Table before they are added to the XLog Queue.
4. After the XLogs are used, the memory that was assigned to them is returned to the XLog Pool.

The following illustration shows the movement of meta data and XLogs within the XLog Collector:

Figure 1-2 The Structure of the XLog Collector



1.1.3 Features of the Log Analyzer

1.1.3.1 The XLog Sender uses the Replication module.

The SQL statements that are used to manage the XLog Sender are almost the same as those that are used to manage Replication. Additionally, the Replication-related properties also apply to the Log Analyzer.

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

1.1.3.2 Transaction XLogs can be obtained in the order in which transactions are committed.

When creating the XLog Collector, it is possible to specify that transaction XLogs are to be obtained in the order in which transactions are committed. When the XLog Collector is set in this way, the following circumstances apply:

- The XLogs pertaining to a given transaction can be obtained after the COMMIT XLog for the transaction has been received.

  Because savepoint-related XLogs are not necessary, they are thus not provided.

- XLogs for transactions that were rolled back cannot be obtained.

1.1.3.3 The TCP and UNIX Domain sockets are supported for use in transmitting XLogs.

- The UNIX Domain socket can be used only when the XLog Sender and XLog Collector reside on the same machine and the OS is UNIX or Linux.
- A single XLog Sender can use only one type of socket.

1.1.3.4 Conversion to ODBC C data types is supported.

Internal data of ALTIBASE HDB can be converted to corresponding ODBC C data types.

1.1.4 Limitations

Because the XLog Sender uses the Replication module, the following limitations apply:

- Only the SYS user can run the XLog Sender.
- The basic unit of analysis is the database table.
- A table to be analyzed must have a primary key.
- The values in the columns that constitute the primary key of a table to be analyzed cannot be updated.

  However, INSERT and DELETE operations can be performed on the columns that constitute the primary key.
- DDL statements cannot be executed on a table to be analyzed.
- The combined total of XLog Senders and Replication Senders that coexist in a single Altibase database cannot exceed 32.
- The same version of the ALTIBASE HDB Replication protocol must be used for both Replication and the Log Analysis API.

  If more than one XLog Collector is being used within a single process, the Replication protocol version in all associated Altibase databases must be the same as the Replication protocol version being used by the Log Analysis API.

However, the Log Analyzer differs from Replication in the following ways:

- The Log Analyzer can be used with tables having foreign keys.
- Only Lazy Mode is supported for use with the Log Analyzer.
- Replication SYNC is not supported.

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

1.2 How to Use the Log Analysis API

This section explains how to use the Log Analysis API, which is provided for use in client applications.

For complete information on the use of the XLog Sender, please refer to 1.1.1.2 XLog Sender.

1.2.1 Required Files

Table 1-1 Required Files

File Type	Filename	Description
Header	alaAPI.h	This file must be included when authorizing a client program using the Log Analysis API. It includes the alaTypes.h file.
	alaTypes.h	This file contains the definitions of data types and macros that are necessary when writing a client program using the Log Analysis API.
Library	libala_sl.x	This is the Log Analysis API shared library.
	libala.x	This is the Log Analysis API static library.

The following must be taken into consideration when creating and compiling source code:

- The source code must reference the alaAPI.h file.

  When creating an application for use in Windows, the windows.h file must be included before the alaAPI.h file in the source code. If the "_WINDOWS_" macro is not defined in the windows.h file, it must be manually defined by the author of the application.

- When linking during the compile operation, it is necessary to link to either the shared library or the static library.

- The filename extension of the library files varies depending on the platform.

1.2.2 Data Types

The basic data types used by the Log Analysis API are as follows:

Table 1-2 Log Analysis API Basic Data Types

Type	Data Type	Description
Boolean	ALA_BOOL	ALA_TRUE: true ALA_FALSE: false
Return Code	ALA_RC	ALA_SUCCESS: success ALA_FAILURE: failure

Type	Data Type	Description
Character	char (SChar)	Signed Character (8 bits)
	unsigned char (UChar)	Unsigned Character (8 bits)
Integer	short (SShort)	Signed Small Integer (16 bits)
	unsigned short (UShort)	Unsigned Small Integer (16 bits)
	int (SInt)	Signed Integer (32 bits)
	unsigned int (UInt)	Unsigned Integer (32 bits)
	long (SLong)	Signed Big Integer (64 bits)
	unsigned long (ULong)	Unsigned Big Integer (64 bits)

1.2.3 Error Handling

Every Log Analysis API function takes the so-called "Error Manager" as an argument. If the result of a call to the Log Analysis API is ALA_FAILURE, it will be necessary to determine the cause of the error and take appropriate measures. The error information that is provided via the API consists of the Error Code, Error Level and Error Message.

The structure of the Error Manager is as follows:

```
typedef struct ALA_ErrorMgr
{
  UInt mErrorCode; /* CODE */
  SChar mErrorState[6]; /* STATE */
  SChar mErrorMessage[ALA_MAX_ERROR_MSG_LEN+256];
} ALA_ErrorMgr;
```

The following should be kept in mind when using the Error Manager:

- The entity (e.g. process or thread) that calls the Log Analysis API is responsible for creating and storing the Error Manager.
- The Error Manager only contains information about the most recent error.
- The functions provided in the Log Analysis API for handling errors cannot accept a NULL value for the Error Manager argument.
- If NULL is passed as the Error Manager argument to a Log Analysis API function other than an error-handling function, then any errors that occur will not be logged by the Log Manager.
- Because `mErrorCode` element of the Error Manager structure contains internal data, the error code must be obtained using ALA_GetErrorCode().

The ALA_GetErrorLevel() function is used to retrieve the value of ALA_ErrorLevel, which indicates the error level.

```
typedef enum
{
    ALA_ERROR_FATAL = 0, /* Need to Destroy */
    ALA_ERROR_ABORT,     /* Need to Handshake */
    ALA_ERROR_INFO       /* Information */
} ALA_ErrorLevel;
```

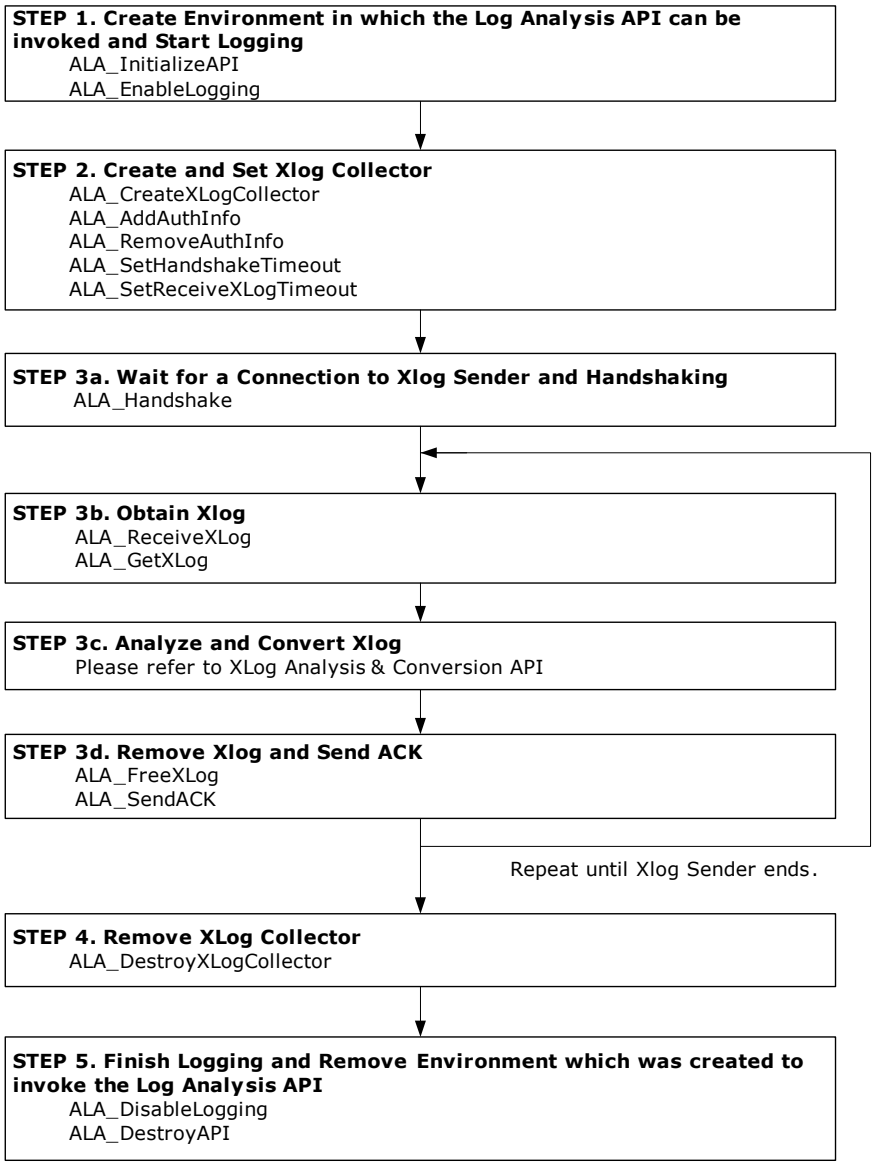
The appropriate action to take in response to an error at each of the supported error levels is as follows:

- ALA_ERROR_FATAL indicates a fatal error, and thus ALA_DestroyXLogCollector() must be called to terminate the corresponding XLog Collector.
- ALA_ERROR_ABORT indicates that the status of the corresponding XLog Collector is abnormal, so ALA_Handshake() must be called to perform handshaking again for the XLog Collector.
- ALA_ERROR_INFO indicates that the call to the Log Analysis API failed. The action that is appropriate in response to this kind of error depends on the error code.

Errors that have already occurred can be checked via the log files written by the Log Manager. For information on the use of the Log Manager, please refer to the descriptions of the ALA_EnableLogging and ALA_DisableLogging.

1.2.4 Basic Use

The following diagram illustrates the steps involved when using the Log Analysis API.



The following must be kept in mind when using the Log Analysis API:

- The XLog Collector can be monitored from Step 3a to Step 3d.
- The Error Handling API is used in all steps.
- When using more than one XLog Collector, Steps 2 to 4 must be performed for each XLog Collector.
- ALA_SendACK() does not have to be called for every iteration of Step 3. For a detailed explanation of when it is necessary to send ACK, please refer to the descriptions of the ALA_CreateXLogCollector and ALA_SendACK.
- If, in Step 3c, the XLog is to be applied to the database using the ODBC interface, AUTOCOMMIT must be set to OFF.

- The XLog Sender must be started after STEP 3a.
- In Step 3b, ALA_ReceiveXLog() and ALA_GetXLog() do not have to be called by the same thread.
- Once ALA_FreeXLog has been called, the corresponding XLog and related data can no longer be used.

1.3 Summary of the Log Analysis API

1.3.1 Log Analysis API Environment Management

Function Type	Log Analysis API Function Name	Description
Creating and Destroying the Log Analysis API Environment	ALA_InitializeAPI	Creates an environment in which the Log Analysis API can be invoked.
	ALA_DestroyAPI	Terminates an environment which was created to invoke the Log Analysis API.
Logging	ALA_EnableLogging	Enables logging for problem tracking.
	ALA_DisableLogging	Disables logging.

1.3.2 XLog Collector Management

Function Type	Log Analysis API Function Name	Description
Creating and Preparing the XLog Collector	ALA_CreateXLogCollector	Creates an XLog Collector that corresponds to an XLog Sender.
	ALA_AddAuthInfo	Adds XLog Sender authentication information.
	ALA_RemoveAuthInfo	Removes XLog Sender authentication information.
	ALA_SetHandshakeTimeout	Specifies the handshake timeout.
	ALA_SetReceiveXLogTimeout	Specifies the XLog reception timeout.
Receiving Meta Data and XLogs	ALA_Handshake	Waits for a connection with an XLog Sender and performs handshaking.
	ALA_ReceiveXLog	Receives XLogs and adds them to the XLog Queue.
	ALA_GetXLog	Obtains an XLog from the XLog Queue.
	ALA_SendACK	Sends ACK to the XLog Sender.
	ALA_FreeXLog	Returns an XLog to the XLog Pool.
Terminating an XLog Collector	ALA_DestroyXLogCollector	Terminates an XLog Collector.

Function Type	Log Analysis API Function Name	Description
Monitoring an XLog Collector	ALA_GetXLogCollectorStatus	Queries the status of an XLog Collector.

1.3.3 XLog Analysis & Conversion

Function Type	Log Analysis API Function Name	Description
Reading XLogs	ALA_GetXLogHeader	Obtains the header information from an XLog.
	ALA_GetXLogPrimaryKey	Obtains the data in the primary key column(s) from an XLog.
	ALA_GetXLogColumn	Obtains the column data (before and after) from an XLog.
	ALA_GetXLogSavepoint	Obtains the savepoint information from an XLog.
	ALA_GetXLogLOB	Obtains the LOB data from an XLog.
Reading meta data	ALA_GetProtocolVersion	Obtains the Replication protocol version of the Log Analysis API.
	ALA_GetReplicationInfo	Obtains Replication information.
	ALA_GetTableInfo	Retrieves information about a table using the table OID.
	ALA_GetTableInfoByName	Retrieves information about a table using the table name and the table owner name.
	ALA_GetColumnInfo	Retrieves information about a column in a table using the column ID.
	ALA_GetIndexInfo	Retrieves information about an index in a table using the index ID.
Reading internal data types of ALTIBASE HDB	ALA_GetInternalNumericInfo	Obtains the sign and exponent of FLOAT or NUMERIC type data.
	ALA_GetAltibaseText	Converts internal data of ALTIBASE HDB into string form.
	ALA_GetAltibaseSQL	Converts an XLog related to a transaction into an SQL string of ALTIBASE HDB.
Type Conversion	ALA_GetODBCCValue	Converts internal data of ALTIBASE HDB into an ODBC C data type.

1.3.4 Error Handling

Function Type	Log Analysis API Function Name	Description
Error Handling	ALA_ClearErrorMgr	Initializes the Error Manager.
	ALA_GetErrorCode	Obtains an error code.
	ALA_GetErrorLevel	Obtains the error level.
	ALA_GetErrorMessage	Obtains a specific error message.

2 The XLog Sender

This chapter explains how to use the XLog Sender, which is one of the components of the ALTIBASE HDB Log Analyzer. The XLog Sender combines log records together to create XLogs, and sends the XLogs to the XLog Collector. The XLog Sender is an internal ALTIBASE HDB module, and is managed in almost exactly the same way that Replication is managed, i.e. using the same SQL interface.

2.1 SQL Statements Used to Manage the XLog Sender

The XLog Sender is managed using ALTIBASE HDB SQL statements that are normally used to manage Replication. The following descriptions highlight only the differences in how these statements are used when they are used to manage the XLog Sender. For complete descriptions of these statements, please refer to the *Replication Manual* and to the *SQL Reference*.

2.1.1 Creating an XLog Sender

This task is accomplished using the CREATE REPLICATION ALTIBASE HDB SQL statement.

2.1.1.1 Syntax

```
CREATE REPLICATION replication_name FOR ANALYSIS
  WITH {{'remote_host_ip', remote_host_ port_no} ... | UNIX_DOMAIN}
  FROM user_name.table_name TO user_name.table_name
  [, FROM user_name.table_name TO user_name.table_name]
  ...;
```

2.1.1.2 Description

This statement is used to create an XLog Sender.

- The XLog Sender is automatically created in LAZY mode. EAGER mode cannot be specified when creating an XLog Sender.
- Unlike Replication, it is possible to specify a UNIX domain connection. This is accomplished by specifying UNIX_DOMAIN in the WITH clause.
- It is acceptable to specify tables with foreign keys in the FROM clause.

With the above exceptions, the use of this statement with an XLog Sender is the same as in Replication.

2.1.1.3 Consideration

The UNIX domain connection protocol can only be used in UNIX and Linux.

2.1.1.4 Example

Create an XLog Sender named *log_analysis* that uses the TCP/IP protocol to send information about table *t1*, which belongs to the *sys* user, to an XLog Collector running on the same machine. The port number must be set to the same port number that is specified in the XLog Collector.

```
iSQL> CREATE REPLICATION log_analysis FOR ANALYSIS
             WITH '127.0.0.1', 20300
             FROM sys.t1 TO sys.t1;
```

2.1.2 Deleting an XLog Sender

This task is accomplished using the DROP REPLICATION ALTIBASE HDB SQL statement.

2.1.2.1 Syntax

```
DROP REPLICATION replication_name;
```

2.1.2.2 Description

The use of this statement with an XLog Sender is the same as in Replication.

2.1.2.3 Example

Delete an XLog Sender named *log_analysis*.

```
iSQL> DROP REPLICATION log_analysis;
```

2.1.3 Starting an XLog Sender

This task is accomplished using the ALTER REPLICATION ALTIBASE HDB SQL statement.

2.1.3.1 Syntax

```
ALTER REPLICATION replication_name {START [AT SN
(xlog_sender_start_sn)]|QUICKSTART};
```

2.1.3.2 Description

This statement is used to start an XLog Sender.

- When using the UNIX domain protocol, the socket filename is automatically generated according to the pattern shown below:

  Socket Filename: $ALTIBASE_HOME/trc/rp-*replication_name*

- Unlike in Replication, an XLog Sender is not registered in the HeartBeat thread when started.
- Unlike the Replication Sender, an XLog Sender can start from the SN specified in the AT SN clause. *xlog_sender_start_sn* is the SN of the XLog from which to start transmission.

With the above exceptions, the use of this statement with an XLog Sender is the same as in Replication.

2.1.3.3 Considerations

- Before the XLog Sender is started, the XLog Collector must be online and waiting for a connection.
- When using the UNIX domain protocol, the setting for the $ALTIBASE_HOME environment

variable must be the same as for the XLog Collector.

- Because the maximum allowable socket filename length varies depending on the operating system, be sure to check and avoid exceeding the maximum allowable length on your system.

In order to start the XLog Sender with the AT SN clause, the following conditions must be satisfied.

- The database must be running in Archivelog mode.
- The number of Log File Groups (LFG) is one, that is, the LOG_FILE_GROUP_COUNT property must be set to 1.
- The dedicated replication log buffer must be not used. That is, the REPLICATION_LOG_BUFFER_SIZE property must be set to 0.

2.1.3.4 Example

Start an XLog Sender named *log_analysis* at the point at which it was last stopped.

```
iSQL> ALTER REPLICATION log_analysis START;
```

2.1.4 Stopping an XLog Sender

This task is accomplished using the ALTER REPLICATION ALTIBASE HDB SQL statement.

2.1.4.1 Syntax

```
ALTER REPLICATION replication_name STOP;
```

2.1.4.2 Description

The use of this statement with an XLog Sender is the same as in Replication.

2.1.4.3 Example

Stop an XLog Sender named *log_analysis*.

```
iSQL> ALTER REPLICATION log_analysis STOP;
```

2.1.5 Adding a Table to a Log Analysis Task

This task is accomplished using the ALTER REPLICATION ALTIBASE HDB SQL statement.

2.1.5.1 Syntax

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

2.1.5.2 Description

This statement is used to add a table to an analysis task.

It is acceptable to specify a table with a foreign key in the FROM clause.

With the above exception, the use of this statement with an XLog Sender is the same as in Replication.

2.1.5.3 Example

Add table *t2*, which belongs to the *sys* user, to the list of tables to be processed by the XLog Sender named *log_analysis*.

```
iSQL> ALTER REPLICATION log_analysis ADD TABLE
 FROM sys.t2 TO sys.t2;
```

2.1.6 Removing a Table from a Log Analysis Task

This task is accomplished using the ALTER REPLICATION ALTIBASE HDB SQL statement.

2.1.6.1 Syntax

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

2.1.6.2 Description

The use of this statement with an XLog Sender is the same as in Replication.

2.1.6.3 Example

Remove table *t2*, which belongs to the *sys* user, from the list of tables to be processed by the XLog Sender named *log_analysis*.

```
iSQL> ALTER REPLICATION log_analysis DROP TABLE
 FROM sys.t2 TO sys.t2;
```

2.1.7 Adding a Host

This task is accomplished using the ALTER REPLICATION ALTIBASE HDB SQL statement.

2.1.7.1 Syntax

```
ALTER REPLICATION replication_name
  ADD HOST 'remote_host_ip', remote_port_no;
```

2.1.7.2 Description

The use of this statement with an XLog Sender is the same as in Replication.

2.1.7.3 Considerations

It is impossible to add a host when the UNIX domain protocol has been specified as the connection type. Furthermore, when adding a host, only TCP/IP can be specified for the new host.

2.1.7.4 Example

Add a host (having the IP address 127.0.0.1 and port number 30301) to an XLog Sender named *log_analysis*.

```
iSQL> ALTER REPLICATION log_analysis
 ADD HOST '127.0.0.1', 30301;
```

2.1.8 Removing a Host

This task is accomplished using the ALTER REPLICATION ALTIBASE HDB SQL statement.

2.1.8.1 Syntax

```
ALTER REPLICATION replication_name
 DROP HOST 'remote_host_ip', remote_port_no;
```

2.1.8.2 Description

The use of this statement with an XLog Sender is the same as in Replication.

2.1.8.3 Consideration

It is only possible to remove a host for which a TCP/IP type connection has been specified.

2.1.8.4 Example

Remove a host (having the IP address 127.0.0.1 and port number 30301) from an XLog Sender named *log_analysis*.

```
iSQL> ALTER REPLICATION log_analysis
 DROP HOST '127.0.0.1', 30301;
```

2.1.9 Setting a Host

This task is accomplished using the ALTER REPLICATION ALTIBASE HDB SQL statement.

2.1.9.1 Syntax

```
ALTER REPLICATION replication_name
    SET HOST 'remote_host_ip', remote_port_no;
```

2.1.9.2 Description

The use of this statement with an XLog Sender is the same as in Replication.

2.1.9.3 Considerations

- The newly set host will be used after the XLog Sender is restarted.
- Only a host for which the connection type is TCP/IP can be specified.

2.1.9.4 Example

Choose the XLog Collector having the IP address 127.0.0.1 and port number 20300 as the active host for an XLog Sender named *log_analysis*.

```
iSQL> ALTER REPLICATION log_analysis
 SET HOST '127.0.0.1', 20300;
```

2.1.10 Flushing XLogs

This task is accomplished using the ALTER REPLICATION ALTIBASE HDB SQL statement.

2.1.10.1 Syntax

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

2.1.10.2 Description

The use of this statement with an XLog Sender is the same as in Replication.

2.1.10.3 Consideration

If the XLog Collector does not send ACK, a timeout may occur.

2.1.10.4 Example

Flush the logs, up to the log that was current at the time point at which the flush command was executed, for the XLog Sender named *log_analysis*. Specify a 10-second timeout.

```
iSQL> ALTER REPLICATION log_analysis FLUSH WAIT 10;
```

2.2 Meta Tables

The meta tables that are used to query the status of Replication objects are also used to query the status of the XLog Sender. For complete descriptions of these meta tables, please refer to the *ALTIBASE HDB General Reference*.

2.2.1 SYSTEM_.SYS_REPLICATIONS_

This meta table contains information on the settings and status of the XLog Sender. A value of 1 in the ROLE column indicates that the Replication object is an XLog Sender.

2.2.2 SYSTEM_.SYS_REPL_HOSTS_

This meta table contains information on the XLog Collector to which the XLog Sender will connect.

For a host that was specified using the UNIX domain protocol, the value in the HOST_IP column is "UNIX_DOMAIN", and the value in the PORT_NO column is the same as the value in the HOST_NO column.

2.2.3 SYSTEM_.SYS_REPL_ITEMS_

This meta table contains information on the tables for which the XLog Sender sends XLogs.

2.3 Performance Views

The performance views that are used to query the status of Replication objects are also used to query the status of the XLog Sender. For complete descriptions of these performance views, please refer to the *ALTIBASE HDB General Reference*.

2.3.1 V$REPEXEC

This performance view displays information about the Replication port and the use of multiple Replication threads.

2.3.2 V$REPSENDER

This performance view displays information on the status of the XLog Sender.

When using the UNIX domain protocol, the value in the SENDER_IP and PEER_IP columns is "UNIX_DOMAIN" and the value in the SENDER_PORT and PEER_PORT columns is 0.

2.3.3 V$REPSENDER_TRANSTBL

This performance view displays information about transaction tables.

2.3.4 V$REPGAP

This performance view displays information about which log file is currently being read by the XLog Sender.

3 Analyzing XLogs

This chapter describes XLogs, meta data and the ALTIBASE HDB internal data types, all of which are required in order to analyze XLogs. XLogs and meta data can be accessed using the Log Analysis API.

3.1 XLogs

This section describes the various kinds of XLogs and their constituent elements.

In order to access XLogs, it is necessary to call ALA_GetXLog().

3.1.1 Types of XLog

```
typedef enum
{
  XLOG_TYPE_BEGIN = 1,               /* Transaction Begin */
  XLOG_TYPE_COMMIT = 2,              /* Transaction Commit */
  XLOG_TYPE_ABORT = 3,               /* Transaction Rollback */
  XLOG_TYPE_INSERT = 4,              /* DML: Insert */
  XLOG_TYPE_UPDATE = 5,              /* DML: Update */
  XLOG_TYPE_DELETE = 6,              /* DML: Delete */
  XLOG_TYPE_SP_SET = 8,              /* Savepoint Set */
  XLOG_TYPE_SP_ABORT = 9,            /* Abort to savepoint */
  XLOG_TYPE_LOB_CURSOR_OPEN = 14,    /* LOB Cursor open */
  XLOG_TYPE_LOB_CURSOR_CLOSE = 15,   /* LOB Cursor close */
  XLOG_TYPE_LOB_PREPARE4WRITE = 16,  /* LOB Prepare for write */
  XLOG_TYPE_LOB_PARTIAL_WRITE = 17,  /* LOB Partial write */
  XLOG_TYPE_LOB_FINISH2WRITE = 18,   /* LOB Finish to write */
  XLOG_TYPE_KEEP_ALIVE = 19,         /* Keep Alive */
  XLOG_TYPE_REPL_STOP = 21           /* Replication Stop */
} ALA_XLogType;
```

There are 13 kinds of transaction-related XLogs and two kinds of control-related XLogs.

The first XLog for a given transaction is XLOG_TYPE_BEGIN, and the last is either XLOG_TYPE_COMMIT or XLOG_TYPE_ABORT.

Because LOB data are typically voluminous, a LOB update task can be associated with more than one XLog. In such cases, the LOB XLogs are received in the sequence shown below:

```
XLOG_TYPE_LOB_CURSOR_OPEN
{
  XLOG_TYPE_LOB_PREPARE4WRITE
  {
    XLOG_TYPE_LOB_PARTIAL_WRITE
    ...
  }
  XLOG_TYPE_LOB_FINISH2WRITE
  ...
}
XLOG_TYPE_LOB_CURSOR_CLOSE
```

The control-related XLogs are KEEP_ALIVE and REPL_STOP.

The KEEP_ALIVE XLog is sent by the XLog Sender to check whether the network connection is still valid when it has no XLogs to send.

The REPL_STOP XLog indicates that the XLog Sender is shutting down normally. After ALA_SendACK() is called, the connection is terminated.

3.1.2 XLog Structure

```
typedef UInt ALA_TID;       /* Transaction ID */
typedef ULong ALA_SN;       /* Log Record SN */
typedef struct ALA_Value    /* Altibase Internal Data */
{
 UInt length;              /* Length of value */
 const void * value;
} ALA_Value;
```

Structure Member	Description
length	The length of the internal ALTIBASE HDB data value
value	The internal ALTIBASE HDB data value

```
typedef struct ALA_XLogHeader             /* XLog Header */
{
 ALA_XLogType mType;                      /* XLog Type */
 ALA_TID mTID;                            /* Transaction ID */
 ALA_SN mSN;                              /* SN */
 ALA_SN mSyncSN;                          /* Reserved */
 ULong mTableOID;                         /* Table OID */
} ALA_XLogHeader;

typedef struct ALA_XLogPrimaryKey         /* Primary Key */
{
 UInt mPKColCnt;                          /* Primary Key Column Count */
 ALA_Value *mPKColArray;                  /* Primary Key Column Value Array */
} ALA_XLogPrimaryKey;

typedef struct ALA_XLogColumn            /* Column */
{
 UInt mColCnt;                           /* Column Count */
 UInt *mCIDArray;                        /* Column ID Array */
 ALA_Value *mBColArray;                  /* Before Image Column Value Array */
 ALA_Value *mAColArray;                  /* After Image Column Value Array */
} ALA_XLogColumn;

typedef struct ALA_XLogSavepoint          /* Savepoint */
{
 UInt mSPNameLen;                         /* Savepoint Name Length */
 SChar *mSPName;                          /* Savepoint Name */
} ALA_XLogSavepoint;

typedef struct ALA_XLogLOB                /* LOB */
{
 ULong mLobLocator;                       /* LOB Locator of Altibase */
 UInt mLobColumnID;
 UInt mLobOffset;
 UInt mLobOldSize;
 UInt mLobNewSize;
 UInt mLobPieceLen;
 UChar *mLobPiece;
} ALA_XLogLOB;

typedef struct ALA_XLog                   /* XLog */
```

```
{
  ALA_XLogHeader mHeader;
  ALA_XLogPrimaryKey mPrimaryKey;
  ALA_XLogColumn mColumn;
  ALA_XLogSavepoint mSavepoint;
  ALA_XLogLOB mLOB;
                                          /* Used internally */
  struct ALA_XLog *mPrev;
  struct ALA_XLog *mNext;
} ALA_XLog;
```

An XLog structure consists of a header, a primary key, a column, a savepoint and LOB-related structures.

Each of these elements can be read either by accessing the `ALA_XLog` structure directly or via the XLog Log Analysis API.

`ALA_XLogPrimaryKey` does not actually contain the primary key column ID array values. These values can be accessed as individual `mColumnID` values in the `mPKColumnArray[sIndex]` array element in the `ALA_Table` structure that contains meta data about the table. The meta data, in turn, can be accessed using either ALA_GetTableInfo() or ALA_GetTableInfoByName().

3.1.3 Configuration Based on XLog Type

The type of XLog can be determined by checking the value of the `mType` member of the `ALA_XLogHeader` structure.

3.1.3.1 BEGIN XLog

Header (mType, mTID, mSN, mSyncSN)

3.1.3.2 COMMIT XLog

Header (mType, mTID, mSN, mSyncSN)

3.1.3.3 ABORT XLog

Header (mType, mTID, mSN, mSyncSN)

3.1.3.4 INSERT XLog

Header (mType, mTID, mSN, mSyncSN, mTableOID)

Column (mColCnt, mCIDArray, mAColArray)

3.1.3.5 UPDATE XLog

Header (mType, mTID, mSN, mSyncSN, mTableOID)

Primary Key (mPKColCnt, mPKColArray)

Column (mColCnt, mCIDArray, mBColArray, mAColArray)

3.1.3.6 DELETE XLog

Header (mType, mTID, mSN, mSyncSN, mTableOID)

Primary Key (mPKColCnt, mPKColArray)

3.1.3.7 SP_SET XLog

Header (mType, mTID, mSN, mSyncSN)

Savepoint (mSPNameLen, mSPName)

- If mSPName begins with "$$IMPLICIT", it is an implicit savepoint.
- If mSPName is "$$PSM_SVP", it is a PSM Savepoint.

3.1.3.8 SP_ABORT XLog

Header (mType, mTID, mSN, mSyncSN)

Savepoint (mSPNameLen, mSPName)

- If `mSPName` begins with "$$IMPLICIT", it is an implicit savepoint.
- If `mSPName` is "$$PSM_SVP", it is a PSM Savepoint.

3.1.3.9 LOB_CURSOR_OPEN XLog

Header (mType, mTID, mSN, mSyncSN, mTableOID)

Primary Key (mPKColCnt, mPKColArray)

LOB (mLobLocator, mLobColumnID)

3.1.3.10 LOB_CURSOR_CLOSE XLog

Header (mType, mTID, mSN, mSyncSN)

LOB (mLobLocator)

3.1.3.11 LOB_PREPARE4WRITE XLog

Header (mType, mTID, mSN, mSyncSN)

LOB (mLobLocator, mLobOffset, mLobOldSize, mLobNewSize)

3.1.3.12 LOB_PARTIAL_WRITE XLog

Header (mType, mTID, mSN, mSyncSN)

LOB (mLobLocator, mLobOffset, mLobPieceLen, mLobPiece)

- `mLobOffset` is the position relative to the value of `mLobOffset` in the

LOB_PREPARE4WRITE XLog.

3.1.3.13 LOB_FINISH2WRITE XLog

Header (mType, mTID, mSN, mSyncSN)

LOB (mLobLocator)

3.1.3.14 KEEP_ALIVE XLog

Header (mType, mTID, mSN, mSyncSN)

3.1.3.15 REPL_STOP XLog

Header (mType, mTID, mSN, mSyncSN)

3.2 Meta Data

This section describes how to access the meta data that are used to interpret XLogs.

Before the meta data can be accessed, it is necessary to call ALA_Handshake().

3.2.1 Meta Data Structure

```
typedef struct ALA_ProtocolVersion
{
  UShort mMajor;                              /* Major Version */
  UShort mMinor;                              /* Minor Version */
  UShort mFix;                                /* Fix Version */
} ALA_ProtocolVersion;

typedef struct ALA_Replication
{
  SChar mXLogSenderName[ALA_NAME_LEN];

  /* XLog Sender Name */
  UInt mTableCount;                           /* Table Count */
  ALA_Table *mTableArray;                     /* Table Array */
} ALA_Replication;

typedef struct ALA_Table
{
  ULong mTableOID;                            /* Table OID */
  SChar mFromUserName[ALA_NAME_LEN];          /* (From) User Name */
  SChar mFromTableName[ALA_NAME_LEN];         /* (From) Table Name */
  SChar mToUserName[ALA_NAME_LEN];            /* (To) User Name */
  SChar mToTableName[ALA_NAME_LEN];           /* (To) Table Name */
  UInt mPKIndexID;                            /* Index ID of Primary Key */
  UInt mPKColumnCount;                        /* Primary Key Column Count */
  ALA_Column **mPKColumnArray;                /* Primary Key Column Array */
  UInt mColumnCount;                          /* Column Count */
  ALA_Column *mColumnArray;                   /* Column Array */
  UInt mIndexCount;                           /* Index Count */
  ALA_Index *mIndexArray;                     /* Index Array */
} ALA_Table;

typedef struct ALA_Column
{
  UInt mColumnID;                             /* Column ID */
  SChar mColumnName[ALA_NAME_LEN];            /* Column Name */
  UInt mDataType;                             /* Column information Type */
  UInt mLanguageID;                           /* Column Language ID */
  UInt mSize;                                 /* Column Size */
  SInt mPrecision;                            /* Column Precision */
  SInt mScale;                                /* Column Scale */
 ALA_BOOL mNotNull;                          /* Column Not Null? */
} ALA_Column;

typedef struct ALA_Index
{
  UInt mIndexID;                              /* Index ID */
  SChar mIndexName[ALA_NAME_LEN];             /* Index Name */
  ALA_BOOL mUnique;                           /* Index Unique? */
  UInt mColumnCount;                          /* Index Column Count */
  UInt *mColumnIDArray;                       /* Index Column ID Array */
} ALA_Index;
```

3.2 Meta Data

The meta data include data about the Protocol Version, Replication, tables, columns and indexes.

The `mPKColumnArray` element of the `ALA_Table` structure is an array of `ALA_Column` pointers.

3.3 ALTIBASE HDB Data Types and Internal Structure

This section describes the format in which the data are stored internally for each data type of ALTIBASE HDB. Information about columns, which is stored in the `ALA_Column` structure, can be accessed by calling ALA_GetColumnInfo(), whereas actual column values, which are stored in the `ALA_Value` structure, can be accessed using the XLog Log Analysis API.

The actual column value is stored in the `value` element of the `ALA_Value` structure, whereas the length of the column value is stored in the `length` element of the `ALA_Value` structure. Meanwhile, the type of data stored in the column can be determined by checking the value of `mDataType` in the ALA_Column structure.

Table 3-1 ALTIBASE HDB Data Types

Category	Data Type	Constant
Number	FLOAT	6
	NUMERIC	2
	DOUBLE	8
	REAL	7
	BIGINT	(UInt)-5
	INTEGER	4
	SMALLINT	5
Date/Time	DATE	9
Character/Binary	CHAR	1
	VARCHAR	12
	NCHAR	(UInt)-8
	NVARCHAR	(UInt)-9
	BYTE	20001
	NIBBLE	20002
	BIT	(UInt)-7
	VARBIT	(UInt)-100
	BLOB	30
	CLOB	40
Spatial	GEOMETRY	10003

3.3.1 FLOAT, NUMERIC

3.3.1.1 Internal Structure

The internal data structures of the FLOAT and NUMERIC types are the same.

```
typedef struct mtdNumericType
{
  UChar length;           /* Length of (signExponent + mantissa) */
  UChar signExponent;     /* Sign and Exponent */
  UChar mantissa[1];      /* UChar Array (Base 100) */
} mtdNumericType;
```

To determine the sign and exponent of a FLOAT or NUMERIC value, call ALA_GetInternalNumericInfo(), or check the values of the mtdNumericType structural elements as shown below.

3.3.1.2 Obtaining the sign from mtdNumericType

```
if(signExponent is 128 ~ 255)
{
  Sign = '+';
}
else /* if(signExponent is 0 ~ 127) */
{
  Sign = '-';
}
```

3.3.1.3 Obtaining the exponent from mtdNumericType

This is the exponent of a decimal number.

```
if(signExponent is 128 ~ 255)
{
  Exponent = ((SInt)(signExponent & 0x7F) - 64) * 2
             + ((mantissa[0] < 10) ? -1 : 0);
}
else  /* if(signExponent is 0 ~ 127) */
{
  Exponent = (64 - (SInt)(signExponent & 0x7F)) * 2
             + ((mantissa[0] >= 90) ? -1 : 0);
}
```

3.3.1.4 Obtaining the mantissa from mtdNumericType

The value of each UChar ranges from 0 and 99 inclusive, that is, each UChar is a base 100 number.

The result is a number between 0 and 1.

```
if(Sign is '+')
{
  /* Example : 01 23 45 67 89 -> 0.123456789
  /*           12 34 56 78 99 -> 0.1234567899
   */

  /* mantissa[0] */
  if(mantissa[0] < 10)
```

```
  {
    MantissaStr = mantissa[0];
  }
  else
  {
    MantissaStr = mantissa[0] / 10;
    MantissaStr = MantissaStr + mantissa[0] % 10;
  }

  /* mantissa[1] - mantissa[mLength - 1] */
  for(Index = 1; Index < mLength - 1; Index++)
  {
    MantissaStr = MantissaStr + mantissa[Index] / 10;
    MantissaStr = MantissaStr + mantissa[Index] % 10;
  }
}
else /* if(Sign is '-') */
{
  /* Example : 98 76 54 32 10 -> 0.123456789
  /*           09 87 65 43 21 -> 0.9012345678
   */

  /* mantissa[0] */
  if(mantissa[0] >= 90)
  {
    MantissaStr = MantissaStr + (99 - mantissa[0]);
  }
  else
  {
    MantissaStr = MantissaStr + (99 - mantissa[0]) / 10;
    MantissaStr = MantissaStr + (99 - mantissa[0]) % 10;
  }

  /* mantissa[1] - mantissa[mLength - 1] */
  for(Index = 1; Index < mLength - 1; Index++)
  {
    MantissaStr = MantissaStr + (99 - mantissa[Index]) / 10;
    MantissaStr = MantissaStr + (99 - mantissa[Index]) % 10;
  }
}
```

3.3.2 DOUBLE, REAL, BIGINT, INTEGER, SMALLINT

3.3.2.1 Internal Structure

Each type is mapped to a primitive data type.

```
typedef SDouble mtdDoubleType;                  /* DOUBLE */
typedef SFloat mtdRealType;                     /* REAL */
typedef SLong mtdBigintType;                    /* BIGINT */
typedef SInt mtdIntegerType;                    /* INTEGER */
typedef SShort mtdSmallintType;                 /* SMALLINT */
```

3.3.3 DATE

3.3.3.1 Internal Structure

There is only one internal data type available for handling dates and times.

```
typedef struct mtdDateType
{
  SShort year;                          /* Year(16bit) */
  UShort mon_day_hour;                  /* Not Used(2bit), Month(4bit), */
                                        /* Day(5bit), Hour(5bit) */
  UInt min_sec_mic;                     /* Minute(6bit), Second(6bit), */
                                        /* MicroSec(20bit) */
} mtdDateType;
```

3.3.4 CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE, NIBBLE, BIT, VARBIT, BLOB, CLOB

3.3.4.1 Internal Structure

These data types have similar structures.

```
typedef struct mtdCharType              /* CHAR, VARCHAR */
{
  UShort length;                        /* Length of value */
  UChar value[1];                       /* UChar Array */
} mtdCharType;

typedef struct mtdNcharType {           /*NCHAR, NVARCHAR */
  UShort length;                        /* Length of value */
  UChar value[1];                       /* UChar Array */
} mtdNcharType;

typedef struct mtdByteType              /* BYTE */
{
  UShort length;                        /* Length of value */
  UChar value[1];                       /* UChar Array */
} mtdByteType;

typedef struct mtdNibbleType            /* NIBBLE */
{
  UChar length;                         /* Length of Nibbles */
  UChar value[1];                       /* UChar Array */
} mtdNibbleType;

typedef struct mtdBitType               /* BIT, VARBIT */
{
  UInt length;                          /* Length of Bits */
  UChar value[1];                       /* UChar Array */
} mtdBitType;

typedef struct mtdLobType
{
  UInt length;                          /* Length of value */
  UChar value[1];                       /* UChar Array */
} mtdLobType;

typedef mtdLobType mtdBlobType;         /* BLOB */
typedef mtdLobType mtdClobType;         /* CLOB */
```

The ALA_GetAltibaseText(), ALA_GetAltibaseSQL() and ALA_GetODBCCValue() functions cannot be used with BLOB or CLOB type values.

Valid values for the `length` element of the NIBBLE type are between 0 and 254 inclusive. A length value of 255 indicates a NULL value.

3.3.5 GEOMETRY

3.3.5.1 Internal Structure

For information on the GEOMETRY data structure and how to handle GEOMETRY type data, please refer to the *ALTIBASE HDB Spatial SQL Reference*. The ALA_GetAltibaseText(), ALA_GetAltibaseSQL() and ALA_GetODBCCValue() functions cannot be used with GEOMETRY type values.

3.4 SAVEPOINT

Savepoints are declared to temporarily save the interim results of partially processed transactions.

In ALTIBASE HDB, the following three kinds of savepoints are available for use:

- Implicit Savepoints
- Explicit Savepoints
- PSM Savepoints

Implicit savepoints are used internally and managed in lists. One such list is maintained for each transaction. Implicit savepoints are used for partial rollback, which is performed automatically if the execution of a particular statement fails, so that only that statement, and not the entire transaction, needs to be rolled back.

It is also possible for users to expressly define explicit savepoints, which are also managed in lists that are maintained for individual transactions. (These lists are managed separately from the lists of implicit savepoints noted above.) Please refer to the *SQL Reference* for more details about explicit savepoints.

PSM savepoints are used internally when stored procedures are executed. These savepoints are only maintained during the execution of stored procedures. For more detailed information about stored procedures, please refer to the *Stored Procedures Manual*.

Each kind of savepoint is managed separately, and savepoint xlogs can be processed in applications depending on the circumstances.

3.4.1 Example

```
iSQL> CREATE TABLE T1 (I1 INTEGER PRIMARY KEY);
Create success.

iSQL> INSERT INTO T1 VALUES (2);
1 row inserted.

iSQL> CREATE OR REPLACE PROCEDURE PROC1
         AS
         BEGIN
             INSERT INTO T1 VALUES(1);
             SAVEPOINT EXPLICIT_SP;
             INSERT INTO T1 VALUES(2);
             INSERT INTO T1 VALUES(3);
         END;
         /
Create success.

iSQL> AUTOCOMMIT OFF;
Set autocommit off success.

iSQL> EXEC PROC1;
[ERR-11058 : The row already exists in a unique index.
    0006 :      INSERT INTO T1 VALUES(2);
               ^                        ^
    ]

iSQL> ROLLBACK TO SAVEPOINT EXPLICIT_SP;
```

```
Rollback success.
```

4 Log Analysis API

This chapter describes how to use the Log Analysis API component of the ALTIBASE HDB Log Analyzer. The Log Analysis API is an API that is invoked by a client application. It provides functions for receiving XLogs from an XLog Sender and analyzing them. In the following function descriptions, any argument whose name begins with “aOut” is an output argument. All of the Log Analysis API functions, which are intended for use in the C and C++ languages, are described in detail in this chapter.

4.1 ALA_InitializeAPI

4.1.1 Syntax

```
ALA_RC ALA_InitializeAPI(
    ALA_BOOL       aUseAltibaseODBCDriver,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.1.2 Arguments

Argument	Description
aUseAltibaseODBCDriver	This indicates whether the ODBC driver of ALTIBASE HDB is being used.
aOutErrorMgr	This is an Error Manager structure.

4.1.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.1.4 Description

This creates an environment in which the Log Analysis API can be invoked.

4.1.5 Considerations

- With the exception of ALA_ClearErrorMgr(), no other Log Analyzer API functions can be called before this function.
- If execution of this function fails, it will be impossible to use the Log Analysis API.
- When using the ODBC Driver of ALTIBASE HDB, it is necessary to call SQLAllocEnv() before calling this function.

4.1.6 Related Function

ALA_DestroyAPI

4.1.7 Example

```
#include <sqlcli.h>
```

```
#include <alaAPI.h>
...

/* When the Altibase ODBC driver is not used */
void testAPIEnvironment1()
{
/* Create Log Analysis API Environment */
(void)ALA_InitializeAPI(ALA_FALSE, NULL);

/* Invoke Log Analysis API */
...

/* Remove Log Analysis API Environment */
(void)ALA_DestroyAPI(ALA_FALSE, NULL);
}

/* When the Altibase ODBC driver is used */
void testAPIEnvironment2(ALA_BOOL aUseAltibaseODBCDriver)
{
SQLHENV sEnv = NULL;

/* Create Altibase ODBC Environment */
(void)SQLAllocEnv(&sEnv);

/* Create Log Analysis API Environment */
(void)ALA_InitializeAPI(ALA_TRUE, NULL);

/* Invoke Altibase ODBC API and Log Analysis API */
...

/* Remove Log Analysis API Environment */
(void)ALA_DestroyAPI(ALA_TRUE, NULL);

/* Remove Altibase ODBC Environment */
(void)SQLFreeEnv(sEnv);
}
```

4.2 ALA_DestroyAPI

4.2.1 Syntax

```
ALA_RC ALA_DestroyAPI(
    ALA_BOOL       aUseAltibaseODBCDriver,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.2.2 Arguments

Argument	Description
aUseAltibaseODBCDriver	This indicates whether the ODBC driver of ALTIBASE HDB was being used.
aOutErrorMgr	This is an Error Manager structure.

4.2.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.2.4 Description

This function terminates the environment that was created by calling ALA_InitializeAPI.

4.2.5 Considerations

- Regardless of the result value, it will be impossible to call any of the Log Analysis API functions after this function has been called.
- When using the ODBC Driver of ALTIBASE HDB, this function must be called before SQL-FreeEnv() is called, which is the last step in terminating the Log Analysis API environment.

4.2.6 Related Function

ALA_InitializeAPI

4.2.7 Example

Please refer to ALA_InitializeAPI.

4.3 ALA_EnableLogging

4.3.1 Syntax

```
ALA_RC ALA_EnableLogging(
    const SChar  * aLogDirectory,
    const SChar  * aLogFileName,
    UInt           aFileSize,
    UInt           aMaxFileNumber,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.3.2 Arguments

Argument	Description
aLogDirectory	This is the log directory.
aLogFileName	This is the name of the log file.
aFileSize	This is the size of each log file.
aMaxFileNumber	This is the maximum number of completed log files.
aOutErrorMgr	This is an Error Manager structure.

4.3.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.3.4 Description

This function enables logging for help in tracking the cause of problems.

If this function is not called, logging will not be performed.

If there is no log file having the specified name, a new log file is created. If a log file having the specified name already exists, logs will be appended to the end of this log file.

If the size of this log file reaches *aFileSize*, the log file is renamed and a new log file having the specified name is created. The header of each log file contains a number ranging from 1 to *aMaxFileNumber*. The log file is renamed with reference to this number. For example, if the header of a log file named “analysis.log” contains the number 1 and the size of the log file reaches the size specified in *aFileSize*, the name of the log file is changed from “analysis.log” to “analysis.log-1”, and a new log file, named “analysis.log” and having the number 2 in the header, is created.

The numbers in the log file headers start at 1 and are incremented by 1. If this number reaches *aMaxFileNumber*, the numbering restarts at 1. As a result, only the log file that is currently in use and the

most recent *aMaxFileNumber* log files are kept.

4.3.5 Considerations

- The maximum length of the log directory and file name (including NULL character) is 1024bytes.
- If logging has already been enabled, calling this function will result in an error.
- Calling another Log Analysis API function while execution of this function is underway will cause unexpected behavior.
- If the contents of the log file header are abnormal or unexpected, the log file is deleted and created again.
- Setting the value of *aFileSize* to 0 will allow the log file to grow infinitely large, depending on system resources.

4.3.6 Related Function

ALA_DisableLogging

4.3.7 Example

```
#include <alaAPI.h>
...
void testLogging()
{
/* Create Log Analysis API Environment */
          (void)ALA_InitializeAPI(ALA_FALSE, NULL);


/* Enable logging
 * Log Directory                                  : The current directory
 * Log File Name                                  : analysis.log
 * The Size of Log File Size : 10 MB
 * The Max. Number of Previous Log Files : 10
 */
      (void)ALA_EnableLogging(".",
                              "analysis.log",
                               10 * 1024 * 1024,
                               10,
                               NULL);

      /* Invoke Log Analysis API */
         ...

      /* Disable logging */
         (void)ALA_DisableLogging(NULL);

      /* Remove Log Analysis API Environment */
         (void)ALA_DestroyAPI(ALA_FALSE, NULL);
}
```

4.4 ALA_DisableLogging

4.4.1 Syntax

```
ALA_RC ALA_DisableLogging(
    ALA_ErrorMgr * aOutErrorMgr);
```

4.4.2 Arguments

Argument	Description
aOutErrorMgr	This is an Error Manager structure.

4.4.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.4.4 Description

This function disables logging.

4.4.5 Considerations

- If logging has not been enabled, calling this function will result in an error.
- Calling another Log Analysis API function while execution of this function is underway will cause unexpected behavior.

4.4.6 Related Function

ALA_EnableLogging

4.4.7 Example

Please refer to ALA_EnableLogging.

4.5 ALA_CreateXLogCollector

4.5.1 Syntax

```
ALA_RC ALA_CreateXLogCollector(
    const SChar  * aXLogSenderName,
    const SChar  * aSocketInfo,
    SInt           aXLogPoolSize,
    ALA_BOOL       aUseCommittedTxBuffer,
    UInt           aACKPerXLogCount,
    ALA_Handle   * aOutHandle,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.5.2 Arguments

Argument	Description
aXLogSenderName	This is the name of the corresponding XLog Sender. (length: 1 - 40)
aSocketInfo	This is the socket type (TCP, UNIX Domain)
aXLogPoolSize	This is the maximum size of the XLog Pool. (unit: the number of XLogs; range: from 1 upwards)
aUseCommittedTxBuffer	This indicates whether to obtain transaction XLogs in the order in which they were committed.
aACKPerXLogCount	This is the actual number of XLogs for which ACK will be sent. (range: from 1 upwards)
aOutHandle	This is the handle for the XLog Collector.
aOutErrorMgr	This is an Error Manager structure.

4.5.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.5.4 Description

This function creates an XLog Collector that corresponds to the specified XLog Sender.

The name of the XLog Sender can be a duplicate name, that is, it can be the same name specified when another XLog collector was created.

aSocketInfo is a string having the format

"SOCKET=*socket_type*;IP_STACK=*xlog_ip_stack*;PEER_IP=*xlog_sender_ip*;MY_PORT=*listen_port*", whereby:

- *socket_type* can only be set to one of "TCP" or "UNIX." If it is set to "UNIX," a socket file having the path and name "$ALTIBASE_HOME/trc/rp-*replication_name*" is automatically created.
- *xlog_sender_ip* must be specified when *socket_type* is set to "TCP". The valid length is in the range from 1 to 39 bytes. This is the IP address of the XLog Sender. This information is required in order to authenticate the XLog Sender.
- *listen_port* must be specified when *socket_type* is set to "TCP". The valid range is from 1024 to 65535 (0xFFFF). This is the port number via which the XLog Sender will connect. It is used when ALA_Handshake() is called.
- *xlog_ip_stack* can be specified when *socket_type* is set to "TCP". This indicates the kind of Internet Protocol Stack to be used.

  If it is not specified, an Internet Protocol Stack that supports only IPv4 will be used. This is the default.
  If it is set to 0, an Internet Protocol Stack that supports only IPv4 will be used.
  If it is set to 1, a dual stack (i.e. an Internet Protocol Stack that supports both IPv4 and IPv6) will be used.
  If it is set to 2, an Internet Protocol Stack that supports only IPv6 will be used.

aACKPerXLogCount is used when ALA_SendACK() is called.

4.5.5 Considerations

- The following must be taken into consideration when the transaction XLogs are obtained in the order in which transactions were committed:
  - Because XLogs for a given transaction accumulate in the transaction table until the corresponding COMMIT XLog arrives, the size of the XLog Pool must be set sufficiently large. In other words, more memory space is required.
  - No XLogs for a transaction can be obtained until after the corresponding COMMIT XLog has been received. In other words, the time interval between the time that an XLog arrives and the time that it is actually processed increases. This means that a decrease in performance is highly likely in situations in which individual transactions are used for batch processing.
- If the number of XLogs for which ACK is to be sent out is set to a number greater than 1 (one), ACK may not be sent to the XLog Sender, even when ALA_SendACK() is called.
- When the socket type is set to "TCP", the listen port must not be being used by any other process.
- When the socket type is set to "UNIX", the value of the ALTIBASE_HOME environment variable must be set the same as in the Altibase database to which the XLog Sender belongs. Additionally, since the maximum allowable length of the socket filename varies depending on the operating system, be sure to check, and avoid exceeding, the maximum allowable length on your system.

4.5.6 Related Functions

ALA_AddAuthInfo

ALA_RemoveAuthInfo

ALA_DestroyXLogCollector

4.5.7 Example

```
#include <alaAPI.h>
     …
void testXLogCollectorTCP()
{
  ALA_Handle sHandle;

/* Create XLog Collector that uses TCP
 * XLog Sender Name : log_analysis
 * XLog Sender Authentication Information : IP=127.0.0.1
 * Listening Port                                          : 30300
 * The max. size of XLog Pool                              : 10000
 * Obtain transaction XLog in the order of commit          : Disabled
 * The reference number of XLog for which ACK will be sent out : 100
 */
  (void)ALA_CreateXLogCollector("log_analysis",
                  "SOCKET=TCP;PEER_IP=127.0.0.1;MY_PORT=30300",
                  10000,
                  ALA_FALSE,
                  100,
                  &sHandle,
                  NULL);

  /* Adde XLog Sender Authentication Information */
  (void) ALA_AddAuthInfo(sHandle, "PEER_IP=127.0.0.2", NULL);

  /* Remove XLog Sender Authentication Information */
  (void)ALA_RemoveAuthInfo(sHandle, "PEER_IP=127.0.0.2", NULL);

  /* Invoke Log Analysis API */
  …

  /* Remove XLog Collector */
  (void)ALA_DestroyXLogCollector(sHandle, NULL);
}

void testXLogCollectorUNIX()
{
  ALA_Handle sHandle;

/* Create XLog Collector that uses a UNIX domain
 * XLog Sender Name : log_analysis
 * The max. size of XLog Pool : 20000
 * Obtain transaction XLog in the order of commit : Enabled
 * The reference number of XLog for which ACK will be sent out : 50
 */
  (void)ALA_CreateXLogCollector("log_analysis",
                                "SOCKET=UNIX",
                                20000,
                                ALA_TRUE,
                                50,
                                &sHandle,
```

```
                                NULL);
  /* Invoke Log Analysis API */
  …

  /* Remove XLog Collector */
  (void)ALA_DestroyXLogCollector(sHandle, NULL);
}
```

4.6 ALA_AddAuthInfo

4.6.1 Syntax

```
ALA_RC ALA_AddAuthInfo(
    ALA_Handle     aHandle,
    const SChar  * aAuthInfo,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.6.2 Arguments

Arguments	Description
aHandle	This is the handle of the XLog Collector.
aAuthInfo	This is the XLog Sender authentication information.
aOutErrorMgr	This is an Error Manager structure.

4.6.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.6.4 Description

This function adds authentication information for the XLog Sender.

When the connection type is TCP, *aAuthInfo* is a string having the format "PEER_IP=xlog_sender_ip". The valid length of *xlog_sender_ip* is from 1 to 39 bytes.

4.6.5 Considerations

- This function can only be called when the socket type is set to "TCP".
- Up to 32 different pieces of XLog Sender authentication information can be specified.

4.6.6 Related Functions

ALA_CreateXLogCollector

ALA_RemoveAuthInfo

ALA_Handshake

4.6.7 Example

Please refer to ALA_CreateXLogCollector.

4.7 ALA_RemoveAuthInfo

4.7.1 Syntax

```
ALA_RC ALA_RemoveAuthInfo(
    ALA_Handle     aHandle,
    const SChar  * aAuthInfo,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.7.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aAuthInfo	This is the XLog Sender authentication information.
aOutErrorMgr	This is an Error Manager structure.

4.7.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.7.4 Description

This function removes the specified XLog Sender authentication information.

When the connection type is TCP, *aAuthInfo* is a string having the format "PEER_IP=xlog_sender_ip". The valid length of *xlog_sender_ip* is from 1 to 39 bytes.

4.7.5 Considerations

- This function can only be called when the socket type is set to "TCP".
- At least one piece of XLog Sender authentication information is required.

4.7.6 Related Functions

ALA_CreateXLogCollector

ALA_AddAuthInfo

ALA_Handshake

4.7.7 Example

Please refer to ALA_CreateXLogCollector.

4.8 ALA_SetHandshakeTimeout

4.8.1 Syntax

```
ALA_RC ALA_SetHandshakeTimeout(
    ALA_Handle     aHandle,
    UInt           aSecond,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.8.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aSecond	This is the handshake timeout. (unit: seconds, range: 1 - 0xFFFFFFFE)
aOutErrorMgr	This is an Error Manager structure.

4.8.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.8.4 Description

This function sets the handshake timeout.

The handshake timeout is used when ALA_Handshake() is called.

The default handshake timeout is 600 seconds.

4.8.5 Related Function

ALA_Handshake

4.8.6 Example

Please refer to ALA_Handshake.

4.9 ALA_SetReceiveXLogTimeout

4.9.1 Syntax

```
ALA_RC ALA_SetReceiveXLogTimeout(
    ALA_Handle     aHandle,
    UInt           aSecond,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.9.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aSecond	This is the reception timeout. (unit: seconds, range: 1 - 0xFFFFFFFE)
aOutErrorMgr	This is an Error Manager structure.

4.9.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.9.4 Description

This function sets the XLog reception timeout.

The XLog reception timeout is used when ALA_ReceiveXLog() is called. The default XLog reception timeout is 10 seconds.

4.9.5 Related Function

ALA_ReceiveXLog

4.9.6 Example

Please refer to ALA_Handshake.

4.10 ALA_Handshake

4.10.1 Syntax

```
ALA_RC ALA_Handshake(
    ALA_Handle     aHandle,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.10.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aOutErrorMgr	This is an Error Manager structure.

4.10.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.10.4 Description

This function is used to tell the XLog Collector to listen for the XLog Sender and perform handshaking.

With the exception of user settings and the XLog Pool, all of the data in the XLog Collector are initialized. Meta data about the target tables are received and stored internally.

4.10.5 Considerations

- When the connection type is TCP and the authentication information does not match, handshaking will fail.
- If the peer being connected to is not an XLog Sender, handshaking will fail.
- If a connection with an XLog Sender is not established within the specified handshaking timeout period, a timeout event will occur.
- ALA_ReceiveXLog(), ALA_GetXLog() and ALA_SendACK() must not be called before handshaking is completed.
- Before commencing handshaking, ALA_FreeXLog() must be executed for all XLogs that were obtained using ALA_GetXLog(), in order to ensure that the XLog Pool is not depleted.

4.10.6 Related Functions

ALA_AddAuthInfo

ALA_RemoveAuthInfo

ALA_SetHandshakeTimeout

ALA_ReceiveXLog

ALA_SendACK

ALA_GetReplicationInfo

ALA_GetTableInfo

ALA_GetColumnInfo

ALA_GetIndexInfo

4.10.7 Example

```
#include <alaAPI.h>
…

void testXLogCollector(ALA_Handle aHandle)
{
ALA_XLog                 * sXLog                 = NULL;
ALA_XLogHeader           * sXLogHeader           = NULL;
ALA_XLogCollectorStatus    sXLogCollectorStatus;
ALA_BOOL                   sInsertXLogInQueue    = ALA_FALSE;
ALA_BOOL                   sExitFlag             = ALA_FALSE;

/* Set Handshake Timeout : 600 seconds */
(void)ALA_SetHandshakeTimeout(aHandle, 600, NULL);

/* Set XLog Receive Timeout : 10 seconds */
(void)ALA_SetReceiveXLogTimeout(aHandle, 10, NULL);

/* Listen for XLog Sender and Handshake */
(void)ALA_Handshake(aHandle, NULL);

/* Receive XLog until XLog Sender ends */
while(sExitFlag != ALA_TRUE)
{
/* Receive XLog and add it to XLog Queue */
sInsertXLogInQueue = ALA_FALSE;
while(sInsertXLogInQueue != ALA_TRUE)
{
 (void)ALA_ReceiveXLog(aHandle, &sInsertXLogInQueue, NULL);
}

/* Obtain XLog from XLog Queue.
 * Assuming that transaction XLog is obtained in the order in which records
are logged.
 */
(void)ALA_GetXLog(aHandle, &sXLog, NULL);

/* Analyze and Process XLog */
(void)ALA_GetXLogHeader(sXLog, &sXLogHeader, NULL);
```

```
if(sXLogHeader->mType == XLOG_TYPE_REPL_STOP)
{
sExitFlag = ALA_TRUE;
}
…

/* Send ACK to XLog Sender */
(void)ALA_SendACK(aHandle, NULL);

/* Return XLog to XLog Pool */
(void)ALA_FreeXLog(aHandle, sXLog, NULL);

/* Obtain the Status of XLog Collector */
(void)ALA_GetXLogCollectorStatus(aHandle,
 &sXLogCollectorStatus,
NULL);
}
}
```

4.11 ALA_ReceiveXLog

4.11.1 Syntax

```
ALA_RC ALA_ReceiveXLog(
    ALA_Handle     aHandle,
    ALA_BOOL     * aOutInsertXLogInQueue,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.11.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aOutInsertXLogInQueue	This indicates whether the received XLog was added to the XLog Queue.
aOutErrorMgr	This is an Error Manager structure.

4.11.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.11.4 Description

This function is used to receive an XLog and add it to the XLog Queue.

The memory for an XLog is obtained from the XLog Pool.

When transaction XLogs are obtained in the order in which transactions are committed, all of the XLogs pertaining to a given transaction are stored in the transaction table until the corresponding COMMIT XLog is received.

This function can be called at the same time as ALA_GetXLog().

4.11.5 Considerations

- ALA_Handshake() must be called before this function. If a network error occurs after a successful call to ALA_Handshake(), the call to this function will fail.
- If no XLogs are received within the XLog reception timeout period, a timeout event will occur.
- If there are no XLogs available in the XLog Pool, the call to this function will fail.

- When transaction XLogs are obtained in the order in which transactions are committed, XLogs that are received are not necessarily added to the XLog Queue.
- If a network error occurs or a REPL_STOP XLog is received, it will be necessary to roll back any uncommitted transactions for which XLogs have been obtained, and in connection with which the database contents have been changed.
- Memory for *aOutInsertXLogInQueue* must be allocated in advance.

4.11.6 Related Functions

ALA_SetReceiveXLogTimeout

ALA_Handshake

ALA_GetXLog

ALA_SendACK

ALA_FreeXLog

4.11.7 Example

Please refer to ALA_Handshake.

4.12 ALA_GetXLog

4.12.1 Syntax

```
ALA_RC ALA_GetXLog(
    ALA_Handle        aHandle,
    const ALA_XLog ** aOutXLog,
    ALA_ErrorMgr    * aOutErrorMgr);
```

4.12.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aOutXLog	This is an XLog obtained from the XLog Queue.
aOutErrorMgr	This is an Error Manager structure.

4.12.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.12.4 Description

This function is used to obtain an XLog from the XLog Queue.

It can be called at the same time as ALA_ReceiveXLog().

If there are no XLogs in the XLog Queue, the value of aOutXLog will be NULL.

4.12.5 Consideration

XLogs must be managed by the client application until ALA_FreeXLog() is called.

4.12.6 Related Functions

ALA_ReceiveXLog

ALA_FreeXLog

ALA_GetXLogHeader

ALA_GetXLogPrimaryKey

ALA_GetXLogColumn

ALA_GetXLogSavepoint

ALA_GetXLogLOB

4.12.7 Example

Please refer to ALA_Handshake.

4.13 ALA_SendACK

4.13.1 Syntax

```
ALA_RC ALA_SendACK(
    ALA_Handle     aHandle,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.13.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aOutErrorMgr	This is an Error Manager structure.

4.13.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.13.4 Description

This function sends ACK to the XLog Sender.

ACK is not sent to the XLog Sender immediately at the time that this function is called. Rather, ACK is sent to the XLog Sender some time after this function is called, when the number of successful calls to ALA_GetXLog() exceeds the number of XLogs for which ACK is to be sent out, or when a KEEP_ALIVE or REPL_STOP XLog is received.

When a REPL_STOP XLog is received, the network connection is disconnected.

The ACK message contains the so-called “Restart SN”. The Restart SN is the SN having the lowest value among the following SNs:

- The lowest XLog SN for active transactions that were checked at the time point at which ALA_GetXLog() was most recently called
- If there are no active transactions, the SN of the last XLog that was obtained using ALA_GetXLog()
- When transaction XLogs are obtained in the order in which transactions are committed, the lowest XLog SN for uncommitted transactions that are currently stored in the transaction table.

4.13.5 Considerations

- ACK has an influence on the XLog Sender's meta table and on flushing. If ACK is not sent within the time period specified in the REPLICATION_RECEIVE_TIMEOUT property on the XLog Sender, the XLog Sender drops the network connection.

  Moreover, if ACK is not sent for a long time, the XLog Sender may stop sending XLogs, update the Restart SN with the SN for the most recently recorded log, and resume attempting to send XLogs.

- Because the restart SN on the XLog Sender can be updated after the XLog Sender receives ACK, it is necessary to process all of the XLogs obtained by calling ALA_GetXLog() before calling ALA_SendACK().

- If a network error occurs, the XLog Sender will periodically attempt handshaking. When handshaking succeeds, the XLog Sender will resume sending XLogs, beginning with the XLog having the Restart SN.

4.13.6 Related Functions

ALA_Handshake

ALA_ReceiveXLog

4.13.7 Example

Please refer to ALA_Handshake.

4.14 ALA_FreeXLog

4.14.1 Syntax

```
ALA_RC ALA_FreeXLog(
    ALA_Handle     aHandle,
    ALA_XLog     * aXLog,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.14.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aXLog	This is an XLog that will be returned to the XLog Pool.
aOutErrorMgr	This is an Error Manager structure.

4.14.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.14.4 Description

This function returns an XLog to the XLog Pool.

4.14.5 Consideration

If XLogs obtained using ALA_GetXLog() are not returned to the XLog Pool for an excessively long period of time, the XLog Pool may become depleted.

4.14.6 Related Functions

ALA_ReceiveXLog

ALA_GetXLog

4.14.7 Example

Please refer to ALA_Handshake.

4.15 ALA_DestroyXLogCollector

4.15.1 Syntax

```
ALA_RC ALA_DestroyXLogCollector(
    ALA_Handle     aHandle,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.15.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aOutErrorMgr	This is an Error Manager structure.

4.15.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.15.4 Description

This function terminates an XLog Collector.

4.15.5 Considerations

- Before this function is called, ALA_FreeXLog() must be executed for all XLogs that were obtained by calling ALA_GetXLog().
- Regardless of the result value, any subsequent calls to the Log Analysis API functions in relation to the corresponding XLog Collector will fail.

4.15.6 Related Function

ALA_CreateXLogCollector

4.15.7 Example

Please refer to ALA_CreateXLogCollector.

4.16 ALA_GetXLogCollectorStatus

4.16.1 Syntax

```
ALA_RC ALA_GetXLogCollectorStatus(
    ALA_Handle                 aHandle,
    ALA_XLogCollectorStatus  * aOutXLogCollectorStatus,
    ALA_ErrorMgr             * aOutErrorMgr);
```

4.16.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aOutXLogCollectorStatus	This is a structure in which information about the status of the XLog Collector is stored.
aOutErrorMgr	This is an Error Manager structure.

4.16.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.16.4 Description

This function is used to check the status of the XLog Collector.

The structure for storing information about the status of the XLog Collector is defined as follows:

```
typedef struct ALA_XLogCollectorStatus
{
 SChar mMyIP[ALA_IP_LEN];
    SInt mMyPort;
    SChar mPeerIP[ALA_IP_LEN];
    SInt mPeerPort;
    SChar mSocketFile[ALA_SOCKET_FILENAME_LEN];
    UInt mXLogCountInPool;
    ALA_SN mLastArrivedSN;
    ALA_SN mLastProcessedSN;
    ALA_BOOL mNetworkValid;
} ALA_XLogCollectorStatus;
```

Structure Member	Description
mMyIP	[TCP] This is the IP address of the XLog Collector.

Structure Member	Description
mMyPort	[TCP] This is the port number being used on the XLog Collector.
mPeerIP	[TCP] This is the IP address of the XLog Sender.
mPeerPort	[TCP] This is the port number being used on the XLog Sender.
mSocketFile	[UNIX Domain] This is the name of the socket file.
mXLogCountInPool	This is the number of XLogs remaining in the XLog Pool.
mLastArrivedSN	This is the SN of the most recently received XLog.
mLastProcessedSN	This is the SN of the most recently processed XLog.
mNetworkValid	This indicates whether the network connection is valid.

These status values are maintained internally by the Log Analyzer, and the values are returned in the structure when this function is called.

The internal values corresponding to mMyIP, mMyPort, mPeerIP, mPeerPort and mSocketFile are updated when ALA_Handshake() is called.

When ALA_ReceiveXLog() is called, the value that is returned in mXLogCountInPool is incremented if a ROLLBACK XLog is received and decremented when other kinds of XLogs are received. It is incremented when ALA_FreeXLog() is called.

`mLastArrivedSN` is the SN of the XLog that was most recently received by calling ALA_ReceiveXLog(). `mLastProcessedSN` is the SN of the XLog that was most recently obtained by calling ALA_GetXLog().

The status value corresponding to `mNetworkValid` might change when ALA_Handshake(), ALA_ReceiveXLog() or ALA_SendACK() is called.

4.16.5 Consideration

Memory for aOutXLogCollectorStatus must be allocated before this function is called.

4.16.6 Related Functions

ALA_Handshake

ALA_ReceiveXLog

ALA_GetXLog

ALA_SendACK

ALA_FreeXLog

4.16.7 Example

Please refer to ALA_Handshake.

4.17 ALA_GetXLogHeader

4.17.1 Syntax

```
ALA_RC ALA_GetXLogHeader(
    const ALA_XLog        * aXLog,
    const ALA_XLogHeader ** aOutXLogHeader,
    ALA_ErrorMgr          * aOutErrorMgr);
```

4.17.2 Arguments

Argument	Description
aXLog	This is the XLog for which to retrieve the header.
aOutXLogHeader	The XLog header of *aXLog* is returned in this argument.
aOutErrorMgr	This is an Error Manager structure.

4.17.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.17.4 Description

This function is used to obtain the XLog header for the specified XLog.

4.17.5 Related Functions

ALA_GetXLog

ALA_GetXLogPrimaryKey

ALA_GetXLogColumn

ALA_GetXLogSavepoint

ALA_GetXLogLOB

4.17.6 Example

```
#include <alaAPI.h>
…
```

```
void testXLogGetPart(const ALA_XLog * aXLog)
{
     ALA_XLogHeader                 * sXLogHeader = NULL;
     ALA_XLogPrimaryKey             * sXLogPrimaryKey = NULL;
     ALA_XLogColumn                 * sXLogColumn = NULL;
     ALA_XLogSavepoint              * sXLogSavepoint = NULL;
     ALA_XLogLOB                    * sXLogLOB = NULL;

     /* Obtain XLog Header */
      (void)ALA_GetXLogHeader(aXLog, &sXLogHeader, NULL);

     /* Obtain XLog Primary Key */
      (void)ALA_GetXLogPrimaryKey(aXLog, &sXLogPrimaryKey, NULL);

     /* Obtain XLog Column */
      (void)ALA_GetXLogColumn(aXLog, &sXLogColumn, NULL);

     /* Obtain XLog Savepoint */
      (void)ALA_GetXLogSavepoint(aXLog, &sXLogSavepoint, NULL);

     /* Obtain XLog LOB */
      (void)ALA_GetXLogLOB(aXLog, &sXLogLOB, NULL);
}
```

4.18 ALA_GetXLogPrimaryKey

4.18.1 Syntax

```
ALA_RC ALA_GetXLogPrimaryKey(
    const ALA_XLog            * aXLog,
    const ALA_XLogPrimaryKey ** aOutXLogPrimaryKey,
    ALA_ErrorMgr              * aOutErrorMgr);
```

4.18.2 Arguments

Argument	Description
aXLog	This is the XLog for which to retrieve the primary key.
aOutXLogPrimaryKey	The XLog primary key of *aXLog* is returned in this argument.
aOutErrorMgr	This is an Error Manager structure.

4.18.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.18.4 Description

This function is used to obtain the XLog primary key for the specified XLog.

4.18.5 Related Functions

ALA_GetXLog

ALA_GetXLogHeader

ALA_GetXLogColumn

ALA_GetXLogSavepoint

ALA_GetXLogLOB

4.18.6 Example

Please refer to ALA_GetXLogHeader.

4.19 ALA_GetXLogColumn

4.19.1 Syntax

```
ALA_RC ALA_GetXLogColumn(
    const ALA_XLog        * aXLog,
    const ALA_XLogColumn ** aOutXLogColumn,
    ALA_ErrorMgr          * aOutErrorMgr);
```

4.19.2 Arguments

Argument	Description
aXLog	This is the XLog for which to retrieve the column.
aOutXLogColumn	The XLog column of *aXLog* is returned in this argument.
aOutErrorMgr	This is an Error Manager structure.

4.19.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.19.4 Description

This function is used to obtain the XLog column for the specified XLog.

4.19.5 Related Functions

ALA_GetXLog

ALA_GetXLogHeader

ALA_GetXLogPrimaryKey

ALA_GetXLogSavepoint

ALA_GetXLogLOB

4.19.6 Example

Please refer to ALA_GetXLogHeader.

4.20 ALA_GetXLogSavepoint

4.20.1 Syntax

```
ALA_RC ALA_GetXLogSavepoint(
    const ALA_XLog           * aXLog,
    const ALA_XLogSavepoint ** aOutXLogSavepoint,
    ALA_ErrorMgr             * aOutErrorMgr);
```

4.20.2 Arguments

Argument	Description
aXLog	This is the XLog for which to retrieve the savepoint.
aOutXLogSavepoint	The XLog savepoint information of *aXLog* is returned in this argument.
aOutErrorMgr	This is an Error Manager structure.

4.20.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.20.4 Description

This function is used to obtain the XLog savepoint information for the specified XLog.

4.20.5 Related Functions

ALA_GetXLog

ALA_GetXLogHeader

ALA_GetXLogPrimaryKey

ALA_GetXLogColumn

ALA_GetXLogLOB

4.20.6 Example

Please refer to ALA_GetXLogHeader.

4.21 ALA_GetXLogLOB

4.21.1 Syntax

```
ALA_RC ALA_GetXLogLOB(
    const ALA_XLog     * aXLog,
    const ALA_XLogLOB ** aOutXLogLOB,
    ALA_ErrorMgr       * aOutErrorMgr);
```

4.21.2 Arguments

Argument	Description
aXLog	This is the XLog for which to retrieve the XLog LOB information.
aOutXLogLOB	The XLog LOB information of *aXLog* is returned in this argument.
aOutErrorMgr	This is an Error Manager structure.

4.21.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.21.4 Description

This function is used to obtain the XLog LOB information for the specified XLog.

4.21.5 Related Functions

ALA_GetXLog

ALA_GetXLogHeader

ALA_GetXLogPrimaryKey

ALA_GetXLogColumn

ALA_GetXLogSavepoint

4.21.6 Example

Please refer to ALA_GetXLogHeader.

4.22 ALA_GetProtocolVersion

4.22.1 Syntax

```
ALA_RC ALA_GetProtocolVersion(
    const ALA_ProtocolVersion * aOutProtocolVersion,
    ALA_ErrorMgr              * aOutErrorMgr);
```

4.22.2 Arguments

Argument	Description
aOutProtocolVersion	This is a Protocol Version structure.
aOutErrorMgr	This is an Error Manager structure.

4.22.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.22.4 Description

This function is used to check the protocol version that is being used by the Log Analysis API.

The protocol version can be checked regardless of whether handshaking has been performed.

4.22.5 Consideration

Before this function is executed, it is first necessary to allocated memory to hold the *aOutProtocolVersion* structure.

4.22.6 Example

```
#include <alaAPI.h>
…

void testProtocolVersion()
{
ALA_ProtocolVersion sProtocolVersion;

/* Obtain Protocol Version */
(void)ALA_GetProtocolVersion(&sProtocolVersion, NULL);
}
```

4.23 ALA_GetReplicationInfo

4.23.1 Syntax

```
ALA_RC ALA_GetReplicationInfo(
    ALA_Handle                aHandle,
    const ALA_Replication ** aOutReplication,
    ALA_ErrorMgr           * aOutErrorMgr);
```

4.23.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aOutReplication	This is information about Replication.
aOutErrorMgr	This is an Error Manager structure.

4.23.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.23.4 Description

This function is used to obtain information about Replication.

4.23.5 Considerations

- If there are no meta data pertaining to replication, this function returns NULL for the *aOutReplication* argument.
- Any meta data that were obtained using this function before handshaking was performed should not be used again after handshaking, as the information might be out of date.

4.23.6 Related Functions

ALA_Handshake

ALA_GetTableInfo

ALA_GetTableInfoByName

ALA_GetColumnInfo

ALA_GetIndexInfo

4.23.7 Example

```
#include <alaAPI.h>
…

void testMetaInformation(ALA_Handle aHandle)
{
ALA_Replication * sReplication      = NULL;
ALA_Table       * sTable            = NULL;
ALA_Table       * sTableByTableOID  = NULL;
ALA_Table       * sTableByName      = NULL;
ALA_Column      * sPKColumn         = NULL;
ALA_Column      * sColumn           = NULL;
ALA_Index       * sIndex            = NULL;
UInt sTablePos;
UInt sPKColumnPos;
UInt sColumnPos;
UInt sIndexPos;

/* Obtain replication information */
(void)ALA_GetReplicationInfo(aHandle, &sReplication, NULL);
for(sTablePos = 0; sTablePos &lt; sReplication->mTableCount; sTablePos++)
{
sTable = &(sReplication->mTableArray[sTablePos]);

/* Obtain table information by table OID */
(void)ALA_GetTableInfo(aHandle,
sTable->mTableOID,
&sTableByTableOID,
NULL);

if(sTableByTableOID != sTable)
{
/* Fatal Error : Error in Log Analysis API */
break;
}

/* Obtain table information by name */
(void)ALA_GetTableInfoByName(aHandle,
sTable->mFromUserName,
sTable->mFromTableName,
&sTableByName,
NULL);

if(sTableByName != sTable)
{
/* Fatal Error : Error in Log Analysis API */
break;
}

/* Process primary key column */
for(sPKColumnPos = 0; sPKColumnPos < sTable->mPKColumnCount; sPKColumnPos++)
{
/* Obtain the primary key column information by primary key column ID */

(void)ALA_GetColumnInfo(sTable,
sTable->mPKColumnArray[sPKColumnPos]->mColumnID,
&sPKColumn,
```

```
NULL);

if(sPKColumn != sTable->mPKColumnArray[sPKColumnPos])
{
/* Fatal Error : Error in Log Analysis API */
break;
}

/* Process primary key column */
…
}

/* Process column */

for(sColumnPos = 0; sColumnPos < sTable->mColumnCount; sColumnPos++)
{
/* Obtain column information by column ID */
(void)ALA_GetColumnInfo(sTable,
sTable->mColumnArray[sColumnPos].mColumnID,
&sColumn,
NULL);
if(sColumn != &(sTable->mColumnArray[sColumnPos]))
{
/* Fatal Error : Error in Log Analysis API */
break;
}

/* Process column */
…
}

/* Process Index */
for(sIndexPos = 0; sIndexPos < sTable->mIndexCount; sIndexPos++)
{
   /* Obtain index information by index ID */
   (void)ALA_GetIndexInfo(sTable,
sTable->mIndexArray[sIndexPos].mIndexID,
&sIndex,
NULL);

 if(sIndex != &(sTable->mIndexArray[sIndexPos]))
 {
/* Fatal Error : Error in Log Analysis API */
break;
 }

 /* Process index */
 …
}
}
}
```

4.24 ALA_GetTableInfo

4.24.1 Syntax

```
ALA_RC ALA_GetTableInfo(
    ALA_Handle          aHandle,
    ULong               aTableOID,
    const ALA_Table  ** aOutTable,
    ALA_ErrorMgr      * aOutErrorMgr);
```

4.24.2 Arguments

Argument	Description
aHandle	This is the handle of the XLog Collector.
aTableOID	This is the OID of the table for which to retrieve the information.
aOutTable	This is information about the table.
aOutErrorMgr	This is an Error Manager structure.

4.24.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.24.4 Description

This function retrieves information about the table identified by the specified table OID.

4.24.5 Considerations

- If there are no meta data corresponding to the specified table, or if the specified table could not be found, this function returns NULL for the *aOutTable* argument.
- Any meta data that were obtained using this function before handshaking was performed should not be used again after handshaking, as the information might be out of date.

4.24.6 Related Functions

ALA_Handshake

ALA_GetReplicationInfo

ALA_GetTableInfoByName

ALA_GetColumnInfo

ALA_GetIndexInfo

4.24.7 Example

Please refer to ALA_GetReplicationInfo.

4.25 ALA_GetTableInfoByName

4.25.1 Syntax

```
ALA_RC ALA_GetTableInfoByName(
    ALA_Handle         aHandle,
    const SChar      * aFromUserName,
    const SChar      * aFromTableName,
    const ALA_Table ** aOutTable,
    ALA_ErrorMgr     * aOutErrorMgr);
```

4.25.2 Arguments

Arguments	Description
aHandle	This is the handle of the XLog Collector.
aFromUserName	This is the name of the owner of the table for which to retrieve the information.
aFromTableName	This is the name of the table for which to retrieve the information.
aOutTable	The table information is returned in this argument.
aOutErrorMgr	This is an Error Manager structure.

4.25.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.25.4 Description

This function retrieves information about the table identified by the specified user (table owner) name and table name.

4.25.5 Considerations

- If there are no meta data corresponding to the specified table, or if no table corresponding to the specified user name and table name could be found, this function returns NULL for the *aOutTable* argument.
- Any meta data that were obtained using this function before handshaking was performed should not be used again after handshaking, as the information might be out of date.

4.25.6 Related Functions

ALA_Handshake

ALA_GetReplicationInfo

ALA_GetTableInfo

ALA_GetColumnInfo

ALA_GetIndexInfo

4.25.7 Example

Please refer to ALA_GetReplicationInfo.

4.26 ALA_GetColumnInfo

4.26.1 Syntax

```
ALA_RC ALA_GetColumnInfo(
    const ALA_Table   * aTable,
    UInt                aColumnID,
    const ALA_Column ** aOutColumn,
    ALA_ErrorMgr      * aOutErrorMgr);
```

4.26.2 Arguments

Argument	Description
aTable	This is the meta information about the table containing the column for which to retrieve the information.
aColumnID	This is the ID of the column for which to retrieve the information.
aOutColumn	The column information is returned in this argument.
aOutErrorMgr	This is an Error Manager structure.

4.26.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.26.4 Description

This function retrieves the information about the column identified by the specified Column ID in the specified table.

4.26.5 Considerations

- If there are no meta data corresponding to the specified column or the specified table, this function returns NULL for the *aOutColumn* argument.
- Any meta data that were obtained using this function before handshaking was performed should not be used again after handshaking, as the information might be out of date.

4.26.6 Related Functions

ALA_Handshake

ALA_GetReplicationInfo

ALA_GetTableInfo

ALA_GetTableInfoByName

ALA_GetIndexInfo

4.26.7 Example

Please refer to ALA_GetReplicationInfo.

4.27 ALA_GetIndexInfo

4.27.1 Syntax

```
ALA_RC ALA_GetIndexInfo(
    const ALA_Table  * aTable,
    UInt               aIndexID,
    const ALA_Index ** aOutIndex,
    ALA_ErrorMgr     * aOutErrorMgr);
```

4.27.2 Arguments

Argument	Description
aTable	This is the meta information about the table containing the index for which to retrieve the information.
aIndexID	This is the ID of the index for which to retrieve the information.
aOutIndex	The index information is returned in this argument.
aOutErrorMgr	This is an Error Manager structure.

4.27.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.27.4 Description

This function retrieves the information about the index identified by the specified Index ID in the specified table.

4.27.5 Considerations

- If there are no meta data corresponding to the specified index, or if the specified table or index could not be found, this function returns NULL for the *aOutIndex* argument.
- Any meta data that were obtained using this function before handshaking was performed should not be used again after handshaking, as the information might be out of date.

4.27.6 Related Functions

ALA_Handshake

ALA_GetReplicationInfo

ALA_GetTableInfo

ALA_GetTableInfoByName

ALA_GetColumnInfo

4.27.7 Example

Please refer to ALA_GetReplicationInfo.

4.28 ALA_GetInternalNumericInfo

4.28.1 Syntax

```
ALA_RC ALA_GetInternalNumericInfo(
    ALA_Column   * aColumn,
    ALA_Value    * aAltibaseValue,
    SInt         * aOutSign,
    SInt         * aOutExponent,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.28.2 Arguments

Argument	Description
aColumn	This is the meta information about the column in which *aAltibaseValue* is stored.
aAltibaseValue	This is the actual ALTIBASE HDB numeric data for which to return the information.
aOutSign	This is the sign of the numeric data.
aOutExponent	This is the exponent of the numeric data.
aOutErrorMgr	This is an Error Manager structure.

4.28.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.28.4 Description

This function is used to determine the sign and exponent of a FLOAT or NUMERIC type number.

An *aOutSign* value of 1 represents a positive number, whereas an *aOutSign* value of 0 represents a negative number.

aOutExponent is a base 10 exponent.

4.28.5 Example

```
#include <alaAPI.h>
…

void testInternalNumeric(ALA_Column * aColumn, ALA_Value * aAltibaseValue)
```

```
{
SInt sNumericSign;
SInt sNumericExponent;

     /* Obtain the internal numeric information */
(void)ALA_GetInternalNumericInfo(aColumn,
aAltibaseValue,
&sNumericSign,
&sNumericExponent,
NULL);
}
```

4.29 ALA_GetAltibaseText

4.29.1 Syntax

```
ALA_RC ALA_GetAltibaseText(
    ALA_Column   * aColumn,
    ALA_Value    * aValue,
    UInt           aBufferSize,
    SChar        * aOutBuffer,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.29.2 Arguments

Argument	Description
aColumn	This is the meta information about the column in which *aValue* is stored.
aValue	This is the actual data of ALTIBASE HDB to convert to a string.
aBufferSize	This is the size of the buffer in which the output string will be stored.
aOutBuffer	This is the buffer containing the actual output string.
aOutErrorMgr	This is an Error Manager structure.

4.29.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.29.4 Description

This function converts ALTIBASE HDB data into string form. For date/time type data, the output string format is "YYYY-MM-DD HH:MI:SS.SSSSSS". BIT type data are converted to the BIT'*value*' format, and VARBIT type data are converted to the VARBIT'*value*' format. NCHAR or NVARCHAR type data are always converted to the UCS-2 (UTF-16) character set.

4.29.5 Considerations

In order to convert NCHAR or NVARCHAR type data between different character sets, it is necessary to set the value of the ALTIBASE_ALA_NCHARSET environment variable on the server on which the logs will be analyzed (i.e. on the XLog Collector side) to the value of the national character set on the server from which the data originate (i.e. on the XLog Sender side).

For example, if the character set for NCHAR and NVARCHAR type data is set to UTF8 on the server acting as the data source, execute the following on the server on which the logs will be analyzed:

```
export ALTIBASE_ALA_NCHARSET=UTF8;
```

Then, when the ALA_GetAltibaseText function is executed on NCHAR or NVARCHAR type data, the data written to aOutBuffer are in UCS-2(UTF-16) form.

```
ex) \0031\0020\0020\0020
```

This function cannot be used with BLOB, CLOB or GEOMETRY type data.

4.29.6 Related Function

ALA_GetAltibaseSQL

4.29.7 Example

```
#include <alaAPI.h>
…

void testAltibaseText(ALA_Table * aTable, ALA_XLog * aXLog)
{
SChar      sBuffer[1024];

/* Obtain Altibase SQL */
(void)ALA_GetAltibaseSQL(aTable,
aXLog,
1024,
sBuffer,
NULL);
}
```

4.30 ALA_GetAltibaseSQL

4.30.1 Syntax

```
ALA_RC ALA_GetAltibaseSQL(
    ALA_Table    * aTable,
    ALA_XLog     * aXLog,
    UInt           aBufferSize,
    SChar        * aOutBuffer,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.30.2 Arguments

Argument	Description
aTable	The table information for XLog
aXLog	XLog
aBufferSize	Buffer Size
aOutBuffer	Buffer
aOutErrorMgr	This is an Error Manager structure.

4.30.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.30.4 Description

This function is used to convert information in an XLog into an SQL string of ALTIBASE HDB. For date/time type data, the output string format is "YYYY-MM-DD HH:MI:SS.SSSSSS". BIT type data are converted to the BIT'*value*' format, and VARBIT type data are converted to the VARBIT'*value*' format. NCHAR and NVARCHAR type data are converted to the UNISTR('*value*') format.

When an INSERT, UPDATE or DELETE SQL statement is reconstructed from XLogs, the target table in the SQL string is specified using the `mToUserName` and `mToTableName` elements of the *aTable* structure.

4.30.5 Considerations

- This function cannot be used with BLOB, CLOB or GEOMETRY type data.
- This function may generate SQL that is incompatible with other DBMS products.

4.30.6 Related Function

ALA_GetAltibaseText

4.30.7 Example

```
#include <alaAPI.h>
…

void testAltibaseSQL(ALA_Table * aTable, ALA_XLog * aXLog)
{
ALA_Column  * sColumn;
SChar         sBuffer[1024];
UInt          sPKColumnPos;
UInt          sColumnPos;

/* Process the primary key column */
for(sPKColumnPos = 0;
sPKColumnPos < aXLog->mPrimaryKey.mPKColCnt;
sPKColumnPos++)
{
/* The primary key sequence for the XLog and the primary key sequence for the
table are the same */
sColumn = aTable->mPKColumnArray[sPKColumnPos];

/* Obtain the Altibase text */
(void)ALA_GetAltibaseText(sColumn,
&(aXLog->mPrimaryKey.mPKColArray[sPKColumnPos]),
1024,
sBuffer,
NULL);
}

/* Process the column */
for(sColumnPos = 0; sColumnPos < aXLog->mColumn.mColCnt; sColumnPos++)
{
/* Obtain the column information */
(void)ALA_GetColumnInfo(aTable,
aXLog->mColumn.mCIDArray[sColumnPos],
&sColumn,
NULL);

/* Obtain the Altibase text for the Before Image */
(void)ALA_GetAltibaseText(sColumn,
&(aXLog->mColumn.mBColArray[sColumnPos]),
1024,
sBuffer,
NULL);

/* Obtain the Altibase text for the After Image */
(void)ALA_GetAltibaseText(sColumn,
&(aXLog->mColumn.mAColArray[sColumnPos]),
1024,
sBuffer,
NULL);
}
}
```

4.31 ALA_GetODBCCValue

4.31.1 Syntax

```
ALA_RC ALA_GetODBCCValue(
    ALA_Column   * aColumn,
    ALA_Value    * aAltibaseValue,
    SInt           aODBCCTypeID,
    UInt           aODBCCValueBufferSize,
    void         * aOutODBCCValueBuffer,
    ALA_BOOL     * aOutIsNull,
    UInt         * aOutODBCCValueSize,
    ALA_ErrorMgr * aOutErrorMgr);
```

4.31.2 Arguments

Argument	Description
aColumn	This is meta information about the column in which *aAltibaseValue* is stored.
aAltibaseValue	This is the data in the internal format of ALTIBASE HDB.
aODBCCTypeID	This is the ODBC C type into which the data are to be converted.
aODBCCValueBufferSize	This is the size of the buffer in which to store the results.
aOutODBCCValueBuffer	This is the buffer in which to store the results.
aOutIsNull	This indicates whether the data to be converted are NULL.
aOutODBCCValueSize	This is the size of the ODBC C value resulting from the conversion.
aOutErrorMgr	This is an Error Manager structure.

4.31.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.31.4 Description

This function is used to convert a value in an internal format of ALTIBASE HDB to an ODBC C type.

The type conversions that are supported are shown in the following table:

Table 4-1 ODBC C Conversion

ALTIBASE HDB Data	SQL_C_CHAR	SQL_C_NUMERIC	SQL_C_BIT	SQL_C_STINYINT SQL_C_UTINYINT	SQL_C_SSHORT SQL_C_USHORT	SQL_C_SLONG SQL_C_ULONG	SQL_C_SBIGINT SQL_C_UBIGINT	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_BINARY	SQL_C_TYPE_DATE SQL_C_TYPE_TIME SQL_C_TYPE_TIMESTAMP
ODBC C Value											
FLOAT	O	O	O	O	O	O	O	O	O	O	
NUMERIC	O	O	O	O	O	O	O	O	O	O	
DOUBLE	O	O	O	O	O	O	O	O	O	O	
REAL	O	O	O	O	O	O	O	O	O	O	
BIGINT	O	O	O	O	O	O	O	O	O	O	
INTEGER	O	O	O	O	O	O	O	O	O	O	
SMALLINT	O	O	O	O	O	O	O	O	O	O	
DATE	O									O	O
CHAR	O	O	O	O	O	O	O	O	O	O	O
VARCHAR	O	O	O	O	O	O	O	O	O	O	O
NCHAR	O	O	O	O	O	O	O	O	O	O	O
NVARCHAR	O	O	O	O	O	O	O	O	O	O	O
BYTE	O									O	
NIBBLE	O									O	
BIT	O	O	O	O	O	O	O	O	O	O	
VARBIT	O	O	O	O	O	O	O	O	O	O	

4.31.5 Considerations

- The BLOB, CLOB and GEOMETRY types cannot be converted using this function.
- This function supports ODBC 3.0 and subsequent versions.

4.31.6 Example

```
#include <alaAPI.h>
…
```

```
void testODBCCConversion(ALA_Table * aTable, ALA_XLog * aXLog)
{
    ALA_Column * sColumn;
    SChar sBuffer[1024];
    ALA_BOOL sIsNull;
    UInt sODBCCValueSize;
    UInt sPKColumnPos;

    /* Convert the primary key to SQL_C_CHAR */
    for(sPKColumnPos = 0;
        sPKColumnPos < aXLog->mPrimaryKey.mPKColCnt;
        sPKColumnPos++)
    {
    /* The primary key sequence for XLog and the primary key sequence for the
table are the same */
    sColumn = aTable->mPKColumnArray[sPKColumnPos];
    /* Convert the internal data to SQL_C_CHAR */
    (void)ALA_GetODBCCValue(sColumn,
        &(aXLog->mPrimaryKey.mPKColArray[sPKColumnPos],
        SQL_C_CHAR,
        1024,
        sBuffer,
        &sIsNull,
        &sODBCCValueSize,
        NULL);
    }
}
```

4.32 ALA_ClearErrorMgr

4.32.1 Syntax

```
ALA_RC ALA_ClearErrorMgr(
    ALA_ErrorMgr * aOutErrorMgr);
```

4.32.2 Arguments

Argument	Description
aOutErrorMgr	This is an Error Manager structure.

4.32.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.32.4 Description

This function initializes the Error Manager.

4.32.5 Consideration

Before an Error Manager is used for the first time, it must be initialized using this function.

4.32.6 Related Functions

ALA_GetErrorCode

ALA_GetErrorLevel

ALA_GetErrorMessage

4.32.7 Example

```
#include <alaAPI.h>
…

void testErrorHandling()
{
ALA_ErrorMgr sErrorMgr
UInt sErrorCode;
ALA_ErrorLevel sErrorLevel;
```

```
SChar * sErrorMessage;
/* Initialize Error Manager */
(void)ALA_ClearErrorMgr(&sErrorMgr);

/* Invoking of Log Analysis API fails*/
…

/* Obtain the error code */
(void)ALA_GetErrorCode(&sErrorMgr, &sErrorCode);

/* Obtain the error level */
(void)ALA_GetErrorLevel(&sErrorMgr, &sErrorLevel);

/* Obtain the error message */
(void)ALA_GetErrorMessage(&sErrorMgr, &sErrorMessage);
}
```

4.33 ALA_GetErrorCode

4.33.1 Syntax

```
ALA_RC ALA_GetErrorCode(
    const ALA_ErrorMgr * aErrorMgr,
    UInt               * aOutErrorCode);
```

4.33.2 Arguments

Argument	Description
aErrorMgr	This is an Error Manager structure.
aOutErrorCode	This is the Error Code.

4.33.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.33.4 Description

This function is used to obtain an error code.

An error code is a unique numerical value that identifies the kind of error that has occurred.

4.33.5 Considerations

- It is necessary to allocate memory in which to store *aOutErrorCode* before calling this function.
- Because the `mErrorCode` element of the ALA_ErrorMgr structure contains information that is internally used and thus not easy to parse or understand, it is necessary to use the ALA_GetErrorCode() function in order to obtain the error code.

4.33.6 Related Functions

ALA_ClearErrorMgr

ALA_GetErrorLevel

ALA_GetErrorMessage

4.33.7 Example

Please refer to ALA_ClearErrorMgr.

4.34 ALA_GetErrorLevel

4.34.1 Syntax

```
ALA_RC ALA_GetErrorLevel(
    const ALA_ErrorMgr * aErrorMgr,
    ALA_ErrorLevel     * aOutErrorLevel);
```

4.34.2 Arguments

Argument	Description
aErrorMgr	This is an Error Manager structure.
aOutErrorLevel	This is the error level.

4.34.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.34.4 Description

This function is used to determine the error level from *aErrorMgr*.

Because the ALA_ERROR_FATAL error level indicates a fatal error, if this error level is returned, the corresponding XLog Collector must be terminated. This is accomplished using the ALA_DestroyXLogCollector() function.

The ALA_ERROR_ABORT error level indicates that the state of the XLog Collector is abnormal. If this error level is returned, it will be necessary to call ALA_Handshake() for the corresponding XLog Collector in order to perform handshaking with the XLog Sender again.

An error level of ALA_ERROR_INFO indicates failure to invoke the Log Analysis API. The action that is appropriate to take in response to this error level is determined based on the error code.

4.34.5 Consideration

It will be necessary to allocate memory in which to store aOutErrorLevel before calling this function.

4.34.6 Related Functions

ALA_ClearErrorMgr

ALA_GetErrorCode

ALA_GetErrorMessage

4.34.7 Example

Please refer to ALA_ClearErrorMgr.

4.35 ALA_GetErrorMessage

4.35.1 Syntax

```
ALA_RC ALA_GetErrorMessage(
    const ALA_ErrorMgr  * aErrorMgr,
    const SChar        ** aOutErrorMessage);
```

4.35.2 Arguments

Argument	Description
aErrorMgr	This is an Error Manager structure.
aOutErrorMessage	The error message is returned in this argument.

4.35.3 Possible Result Values

ALA_SUCCESS

ALA_FAILURE

4.35.4 Description

This function is used to obtain the actual error message from an Error Manager.

4.35.5 Consideration

Since only a pointer to the actual error message is returned in *aOutErrorMessage*, there is no need to allocate memory in which to store *aOutErrorMessage*.

4.35.6 Related Functions

ALA_ClearErrorMgr

ALA_GetErrorCode

ALA_GetErrorLevel

4.35.7 Example

Please refer to ALA_ClearErrorMgr.

Appendix A. Error Codes

Error Code Table

The Log Analyzer error codes and the cause of each kind of error are set forth in the following table.

FATAL Errors

Error Code	Description	Can be returned by
0x50008	Returned in response to an attempt to begin a transaction that is already active	ALA_ReceiveXLog ALA_GetXLog
0x5000A	Failed to initialize a mutex	ALA_CreateXLogCollector ALA_Handshake
0x5000B	Failed to remove a mutex	ALA_Handshake ALA_DestroyXLogCollector
0x5000C	Failed to lock a mutex	ALA_AddAuthInfo ALA_RemoveAuthInfo ALA_Handshake ALA_ReceiveXLog ALA_GetXLog ALA_SendACK ALA_FreeXLog ALA_DestroyXLogCollector ALA_GetXLogCollectorStatus
0x5000D	Failed to unlock a mutex	

ABORT Errors

Error Code	Description	Can be returned by
0x51006	Failed to allocate memory	All Log Analysis API functions
0x5101E	Failed to allocate memory in pool	ALA_ReceiveXLog
0x5101F	Failed to free memory in pool	ALA_Handshake ALA_ReceiveXLog ALA_FreeXLog ALA_DestroyXLogCollector
0x51020	Failed to initialize the memory pool	ALA_CreateXLogCollector
0x51021	Failed to remove the memory pool	ALA_DestroyXLogCollector

Error Code	Description	Can be returned by
0x51013	Failed to initialize the network environment	ALA_Handshake ALA_ReceiveXLog ALA_SendACK
0x51019	Failed to remove a network protocol	
0x5101A	Failed to finalize the network environment	
0x51017	The network session has already been terminated.	ALA_ReceiveXLog ALA_SendACK
0x51018	The network protocol is unfamiliar.	ALA_Handshake ALA_ReceiveXLog
0x51016	Failed to read from the network	
0x5101B	Failed to write to the network	ALA_Handshake ALA_SendACK
0x5101C	Failed to flush the network	
0x51015	A network timeout (and thus a probable network error) has occurred.	ALA_Handshake
0x5102C	Failed to add a network session	ALA_Handshake
0x51024	Protocol versions are mismatched.	ALA_Handshake
0x51027	Failed to allocate a link	ALA_Handshake
0x51028	Failed to listen for a link	ALA_Handshake
0x51029	Failed to wait for a link	ALA_Handshake
0x5102A	Failed to accept a link	ALA_Handshake
0x5102B	Failed to set a link	ALA_Handshake
0x51022	Failed to shut down a link	ALA_Handshake ALA_DestroyXLogCollector
0x51023	Failed to free a link	
0x51012	The meta information does not exist.	ALA_Handshake ALA_GetXLog ALA_GetReplicationInfo ALA_GetTableInfo ALA_GetTableInfoByName
0x5103F	The table information does not exist.	ALA_GetXLog
0x51040	The column information does not exist.	ALA_GetXLog

INFO Errors

Error Code	Description	Can be returned by
0x52034	Log Analysis API Environment Create Failed	ALA_InitializeAPI
0x52035	Log Analysis API Environment Remove Failed	ALA_DestroyAPI
0x52000	Log Manager Initialization Failure	ALA_EnableLogging
0x52001	Log File Open Failure	ALA_EnableLogging
0x52004	Log Manager Lock Failure	All Log Analysis API functions
0x52005	Log Manager Unlock Failure	All Log Analysis API functions
0x52003	Log Manager Remove Failure	ALA_DisableLogging
0x52002	Log File Close Failure	ALA_DisableLogging
0x52009	Not an active transaction	ALA_GetXLog
0x5200E	The linked list is not empty.	ALA_Handshake ALA_DestroyXLogCollector
0x52033	XLog Pool is empty.	ALA_ReceiveXLog
0x5200F	NULL Parameter	All Log Analysis API functions
0x5201D	Invalid Parameter	All Log Analysis API functions
0x52014	Network Timeout (can be retried)	ALA_ReceiveXLog
0x52026	A socket type that is not supported.	ALA_Handshake
0x52025	A socket type is not selected.	ALA_Handshake
0x5202F	The socket type does not support the corresponding Log Analysis API.	ALA_AddAuthInfo ALA_RemoveAuthInfo
0x5202D	The XLog Sender name is different.	ALA_Handshake
0x52030	There is only one piece of authentication information available.	ALA_RemoveAuthInfo
0x52031	No more authentication information can be added.	ALA_AddAuthInfo
0x52032	There is no authentication information available for a peer.	ALA_Handshake
0x52010	Invalid Role	ALA_Handshake
0x52011	Invalid Replication Flags	ALA_Handshake
0x52007	Geometry Endian Conversion Failure	ALA_GetXLog

Error Code	Description	Can be returned by
0x52036	Unable to obtain the MTD module.	ALA_GetXLog ALA_GetAltibaseText ALA_GetAltibaseSQL
0x52037	Failed to create text with the MTD module.	ALA_GetAltibaseText
0x52038	CMT Initialization Failure	ALA_GetODBCCValue
0x52039	CMT End Failure	ALA_GetODBCCValue
0x5203A	Analysis Header Create Failed (ODBC Conversion)	ALA_GetODBCCValue
0x5203B	Analysis Header Remove Failure (ODBC Conversion)	ALA_GetODBCCValue
0x5203C	Failed to convert from MT to CMT.	ALA_GetODBCCValue
0x5203D	Failed to convert from CMT to ulnColumn.	ALA_GetODBCCValue
0x5203E	Failed to convert from ulnColumn to ODBC C.	ALA_GetODBCCValue

Appendix B. Sample Code

Sample Code: Replication to Altibase DBMS

This sample code is available at $ALTIBASE_HOME/sample/ALA/Altibase/ReplToAltiSample.c

The following is a brief description of the use of the Log Analyzer with reference to sample code. For complete information, please refer to Chapter2: The XLog Sender for details.

XLog Sender Creation

```
CREATE REPLICATION ALA1 FOR ANALYSIS
    WITH '127.0.0.1', 47146
    FROM ala.ala_t1 TO ala.ala_t1;
```

XLog Collector Execution

```
./ReplToAltiSample
```

XLog Sender Startup

```
ALTER REPLICATION ALA1 START;
```

Sample Code

This sample code is available at $ALTIBASE_HOME/sample/ALA/Altibase/ReplToAltiSample.c

```
/
*************************************************************************
**
 * Replication to Altibase DBMS Sample *
 * based on Committed Transaction Only *

*************************************************************************
**/
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

/* Include Altibase ODBC header */
#include <sqlcli.h>

/* Include Altibase Log Analysis API header */
#include <alaAPI.h>

/* User-specific Definitions */
#define QUERY_SIZE (4196)            /* SQL Query Buffer Size */
```

```
#define ALA_LOG_FILE "ALA1.log"      /* Log File Name */
#define ALA_NAME "ALA1"              /* XLog Sender Name */
#define SOCKET_TYPE "TCP"            /* TCP or UNIX */
#define PEER_IP "127.0.0.1"          /* TCP : XLog Sender IP */
#define MY_PORT (47146)              /* TCP : XLog Collector Listen Port */
#define SLAVE_IP "127.0.0.1"         /* ODBC : Target Altibase DBMS IP */
#define SLAVE_PORT (43146)           /* ODBC : Target Altibase DBMS Port */

/* Get XLog from XLog Sender, after handshake with XLog Sender */
ALA_RC runXLogCollector(ALA_Handle, ALA_ErrorMgr *);

/* And, apply XLog to Altibase DBMS */
ALA_RC applyXLogToAltibase(ALA_Handle, ALA_XLog *, ALA_ErrorMgr *);

/* Print error to console */
void printSqlErr(SQLHDBC, SQLHSTMT);
void printAlaErr(ALA_ErrorMgr * aErrorMgr);

/* ODBC variables */
SQLHENV gEnv;
SQLHDBC gDbc;
SQLHSTMT gStmt;

/* Start function */
int main(void)
{
 ALA_Handle sHandle;                 /* XLog Collector Handle */
 ALA_ErrorMgr sErrorMgr;             /* Error Manager */
 char sSocketInfo[128];       /* XLog Sender/Collector Socket Information */
 SQLCHAR sConInfo[128];              /* ODBC Connection Information */
 unsigned int sStep = 0;

 /*************************************************************************
 * Altibase ODBC Initialization *
 *************************************************************************/

 /* If you call SQLAllocEnv() that is included in Altibase ODBC,
  * you have to set ALA_TRUE to the first parameter
  * when you call ALA_IntializeAPI()
 */
 if(SQLAllocEnv(&gEnv) != SQL_SUCCESS)
 {
 goto FINALYZE;
 }
 sStep = 1;

 if(SQLAllocConnect(gEnv, &gDbc) != SQL_SUCCESS)
 {
 goto FINALYZE;
 }
 sStep = 2;

 memset(sConInfo, 0x00, 128);
 sprintf((char *)sConInfo, "DSN=%s;UID=SYS;PWD=MANAGER;PORT_NO=%d",
 SLAVE_IP,
                              SLAVE_PORT);

 if(SQLDriverConnect(gDbc, NULL, sConInfo, SQL_NTS, NULL, 0, NULL,
SQL_DRIVER_NOPROMPT)
 != SQL_SUCCESS)
 {
 printSqlErr(gDbc, gStmt);
 goto FINALYZE;
 }
 sStep = 3;
```

```
 /* Autocommit OFF */
 if(SQLSetConnectAttr(gDbc, SQL_ATTR_AUTOCOMMIT, (SQL-
POINTER)SQL_AUTOCOMMIT_OFF, 0)
 != SQL_SUCCESS)
 {
 printSqlErr(gDbc, gStmt);
 goto FINALYZE;
 }

 /*************************************************************************
 * ALA Initialization *
 *************************************************************************/

 /* Initialize Error Manager */
 (void)ALA_ClearErrorMgr(&sErrorMgr);

 /* Initialize ALA API environment */
 if(ALA_InitializeAPI(ALA_TRUE, &sErrorMgr) != ALA_SUCCESS)
 {
 printAlaErr(&sErrorMgr);
 goto FINALYZE;
 }
 sStep = 4;

 /* Initialize ALA Logging */
 if(ALA_EnableLogging((const signed char *)".", /* Current Directory */
 (const signed char *)ALA_LOG_FILE,             /* Log File Name */
 10 * 1024 * 1024,                              /* Log File Size */
 20,                                /* Maximum Previous Log File Count */
               &sErrorMgr)
            != ALA_SUCCESS)
 {
       printAlaErr(&sErrorMgr);
       goto FINALYZE;
 }
 sStep = 5;

 /* Create XLogCollector */
 memset(sSocketInfo, 0x00, 128);
 sprintf(sSocketInfo, "SOCKET=%s;PEER_IP=%s;MY_PORT=%d",
                       SOCKET_TYPE,
                       PEER_IP,
                       MY_PORT);
 if(ALA_CreateXLogCollector((const signed char *)ALA_NAME,
                           (const signed char *)sSocketInfo,
                            10000,               /* XLog Pool Size */
                            ALA_TRUE, /* Use Committed Transaction Buffer */
                            100,                 /* ACK Per XLog Count */
                            &sHandle,
                            &sErrorMgr)
              != ALA_SUCCESS)
 {
     printAlaErr(&sErrorMgr);
     goto FINALYZE;
 }
 sStep = 6;

 /* Set Timeouts */
 if(ALA_SetHandshakeTimeout(sHandle, 600, &sErrorMgr) != ALA_SUCCESS)
 {
     printAlaErr(&sErrorMgr);
     goto FINALYZE;
 }
 if(ALA_SetReceiveXLogTimeout(sHandle, 10, &sErrorMgr) != ALA_SUCCESS)
```

```
{
    printAlaErr(&sErrorMgr);
    goto FINALYZE;
}

/*****************************************************************************
* Using XLog Collector *
*****************************************************************************/

(void)runXLogCollector(sHandle, &sErrorMgr);

FINALYZE:
/*****************************************************************************
* Finalization *
*****************************************************************************/

switch(sStep)
{
     case 6:
          /* Destroy XLog Collector */
          (void)ALA_DestroyXLogCollector(sHandle, &sErrorMgr);

     case 5:
          /* Finalize Logging */
          (void)ALA_DisableLogging(&sErrorMgr);

     case 4:
          /* Destroy ALA API environment */
          (void)ALA_DestroyAPI(ALA_TRUE, &sErrorMgr);

     case 3:
          (void)SQLDisconnect(gDbc);

     case 2:
          (void)SQLFreeConnect(gDbc);

     case 1:
          (void)SQLFreeEnv(gEnv);

     default:
               break;
}
return 0;
}
ALA_RC runXLog Collector(ALA_Handle aHandle, ALA_ErrorMgr * aErrorMgr)
{
     ALA_XLog * sXLog = NULL;
     ALA_XLogHeader * sXLogHeader = NULL;
     UInt sErrorCode;
     ALA_ErrorLevel sErrorLevel;
     ALA_BOOL sReplStopFlag = ALA_FALSE;
     ALA_BOOL sDummyFlag = ALA_FALSE;
     ALA_BOOL sAckFlag;

     /* Run until ALA_ERROR_FATAL Error occurs or REPL_STOP XLog arrives */
      while(sReplStopFlag != ALA_TRUE)
       {
         /* Wait and Handshake with XLog Sender */
            if(ALA_Handshake(aHandle, aErrorMgr) != ALA_SUCCESS)
            {
              printAlaErr(aErrorMgr);
              (void)ALA_GetErrorLevel(aErrorMgr, &sErrorLevel);
               if(sErrorLevel == ALA_ERROR_FATAL)
               {
                  return ALA_FAILURE;
```

```
        }
        /* Wait and Handshake with XLog Sender */
        continue;
      }

       while(sReplStopFlag != ALA_TRUE)
      {
        /* Get XLog from XLog Queue */
         if(ALA_GetXLog(aHandle, (const ALA_XLog **)&sXLog, aErrorMgr)
          != ALA_SUCCESS)
                 {
                    printAlaErr(aErrorMgr);
                    (void)ALA_GetErrorLevel(aErrorMgr, &sErrorLevel);
                    if(sErrorLevel == ALA_ERROR_FATAL)
                    {
                       return ALA_FAILURE;
                    }
                    /* Wait and Handshake with XLog Sender */
                     break;
        }
        else
        {
             /* If XLog is NULL, then Receive XLog */
             if(sXLog == NULL)
             {
               /* Receive XLog and Insert into Queue */
                if(ALA_ReceiveXLog(aHandle, &sDummyFlag, aErrorMgr)
!= ALA_SUCCESS)
             {
                printAlaErr(aErrorMgr);
                (void)ALA_GetErrorLevel(aErrorMgr, &sErrorLevel);
                if(sErrorLevel == ALA_ERROR_FATAL)
                {
                    return ALA_FAILURE;
                 }
                else
                {
                    (void)ALA_GetErrorCode(aErrorMgr, &sErrorCode);
                    if(sErrorCode == 0x52014) /* Timeout */
                 {
                    /* Receive XLog and Insert into Queue */
                    continue;
                 }
             }
              /* Wait and Handshake with XLog Sender */
              break;
      }

      /* Get XLog from XLog Queue */
       continue;
      }

      /* Get XLog Header */
      (void)ALA_GetXLogHeader(sXLog,
                              (const ALA_XLogHeader **)&sXLogHeader,
                               aErrorMgr);

       /* Check REPL_STOP XLog */
       if(sXLogHeader->mType == XLOG_TYPE_REPL_STOP)
       {
            sReplStopFlag = ALA_TRUE;
       }

        /* Apply XLog to Altibase DBMS */
        sAckFlag = ALA_FALSE;
```

```
    switch(sXLogHeader->mType)
    {
       se XLOG_TYPE_COMMIT :
       case XLOG_TYPE_ABORT : /* Unused in Committed Transaction
                                 Only */
       case XLOG_TYPE_REPL_STOP :
            (void)applyXLogToAltibase(aHandle, sXLog, aErrorMgr);
            sAckFlag = ALA_TRUE;
            break;

       case XLOG_TYPE_INSERT :
       case XLOG_TYPE_UPDATE :
       case XLOG_TYPE_DELETE :
       case XLOG_TYPE_SP_SET : /* Unused in Committed Transaction
                                 Only */
      case XLOG_TYPE_SP_ABORT : /* Unused in Committed Transaction
                                 Only */
       (void)applyXLogToAltibase(aHandle, sXLog, aErrorMgr);
        break;

       case XLOG_TYPE_KEEP_ALIVE :
            sAckFlag = ALA_TRUE;
            break;

       case XLOG_TYPE_BEGIN :
       case XLOG_TYPE_LOB_CURSOR_OPEN :
       case XLOG_TYPE_LOB_CURSOR_CLOSE :
       case XLOG_TYPE_LOB_PREPARE4WRITE :
       case XLOG_TYPE_LOB_PARTIAL_WRITE :
       case XLOG_TYPE_LOB_FINISH2WRITE :
       default :
           break;
     }

     /* Free XLog */
     if(ALA_FreeXLog(aHandle, sXLog, aErrorMgr) != ALA_SUCCESS)
     {
         printAlaErr(aErrorMgr);
         (void)ALA_GetErrorLevel(aErrorMgr, &sErrorLevel);
         if(sErrorLevel == ALA_ERROR_FATAL)
         {
             return ALA_FAILURE;
         }
         /* Wait and Handshake with XLog Sender */
         break;
     }

     /* Send ACK to XLog Sender */
     if(sAckFlag != ALA_FALSE)
     {
         if(ALA_SendACK(aHandle, aErrorMgr) != ALA_SUCCESS)
         {
              printAlaErr(aErrorMgr);
              (void)ALA_GetErrorLevel(aErrorMgr, &sErrorLevel);
              if(sErrorLevel == ALA_ERROR_FATAL)
              {
                  return ALA_FAILURE;
              }
              /* Wait and Handshake with XLog Sender */
              break;
          }
     }
   } /* else */
 } /* while */
```

```
     /* Rollback Current Transaction */
      (void)SQLEndTran(SQL_HANDLE_DBC, gDbc, SQL_ROLLBACK);
   } /* while */

   return ALA_SUCCESS;
}

ALA_RC applyXLogToAltibase(ALA_Handle aHandle, ALA_XLog * aXLog,
ALA_ErrorMgr * aErrorMgr)
{
    ALA_Table         * sTable = NULL;
    ALA_XLogHeader    * sXLogHeader = NULL;
    char                sQuery[QUERY_SIZE];
    char              * sImplictSPPos;

 /* Get XLog Header */
(void)ALA_GetXLogHeader(aXLog,
                        (const ALA_XLogHeader **)&sXLogHeader,
                         aErrorMgr);

  /* if COMMIT XLog, then Commit Current Transaction */
  if(sXLogHeader->mType == XLOG_TYPE_COMMIT)
  {
       (void)SQLEndTran(SQL_HANDLE_DBC, gDbc, SQL_COMMIT);
  }
  /* if ABORT XLog, then Rollback Current Transaction */
  else if(sXLogHeader->mType == XLOG_TYPE_ABORT)
  {
       (void)SQLEndTran(SQL_HANDLE_DBC, gDbc, SQL_ROLLBACK);
  }
  /* if REPL_STOP XLog, then Rollback Current Transaction */
  else if(sXLogHeader->mType == XLOG_TYPE_REPL_STOP)
  {
       (void)SQLEndTran(SQL_HANDLE_DBC, gDbc, SQL_ROLLBACK);
  }
  /* etc. */
  else
  {
       /* Get Table Information */
       if(ALA_GetTableInfo(aHandle,
                           sXLogHeader->mTableOID,
                          (const ALA_Table **)&sTable,
                           aErrorMgr) != ALA_SUCCESS)
       {
            printAlaErr(aErrorMgr);
            return ALA_FAILURE;
       }

       /* Get Altibase SQL from XLog */
       memset(sQuery, 0x00, QUERY_SIZE);
       if(ALA_GetAltibaseSQL(sTable,
                             aXLog,
                             QUERY_SIZE,
                             (signed char *)sQuery, aErrorMgr)
            != ALA_SUCCESS)
       {
            printAlaErr(aErrorMgr);
            return ALA_FAILURE;
       }

       /* In order to Apply Implicit Savepoint to Altibase DBMS,
        * '$' characters of Savepoint's Name has to be changed.
        * Unused in Committed Transaction Only */
       if((sXLogHeader->mType == XLOG_TYPE_SP_SET) ||
          (sXLogHeader->mType == XLOG_TYPE_SP_ABORT))
```

```
            {
                while((sImplictSPPos = strchr(sQuery, '$')) != NULL)
                {
                    *sImplictSPPos = '_';
                }
            }

            /* Apply SQL to DBMS with ODBC */
            if(SQLAllocStmt(gDbc, &gStmt) != SQL_SUCCESS)
            {
                 return ALA_FAILURE;
            }

            if(SQLExecDirect(gStmt, (SQLCHAR *)sQuery, SQL_NTS) !=
            SQL_SUCCESS)
            {
                 printSqlErr(gDbc, gStmt);
                 (void)SQLFreeStmt(gStmt, SQL_DROP);
                 return ALA_FAILURE;
            }

            (void)SQLFreeStmt(gStmt, SQL_DROP);
        }
    return ALA_SUCCESS;
}

void printSqlErr(SQLHDBC aDbc, SQLHSTMT aStmt)
{
    SQLINTEGER errNo;
    SQLSMALLINT msgLength;
    SQLCHAR errMsg[1024];

    if(SQLError(SQL_NULL_HENV, aDbc, aStmt,
                NULL, &errNo,
                errMsg, sizeof(errMsg), &msgLength)
          == SQL_SUCCESS)
    {
         printf("SQL Error : %d, %s\n", (int)errNo, (char *)errMsg);
    }
}

void printAlaErr(ALA_ErrorMgr * aErrorMgr)
{
    char * sErrorMessage = NULL;
    int sErrorCode;

    (void)ALA_GetErrorCode(aErrorMgr, (unsigned int *)&sErrorCode);
    (void)ALA_GetErrorMessage(aErrorMgr, (const signed char **)&sErrorMes
     sage);

     printf("ALA Error : %d, %s\n", sErrorCode, sErrorMessage);
}
```

Index