Altibase Application Development

Altibase C Interface Manual

Release 6.1.1

April 20, 2012

Altibase Application Development Altibase C Interface Manual

Release 6.1.1

Altibase Corporation

10F, Daerung PostTower II, 182-13,

Guro-dong Guro-gu Seoul, 152-847, Korea

Telephone: +82-2-2082-1000   Fax: 82-2-2082-1099

E-mail: support@altibase.com   www: http://www.altibase.com

Contents

Preface

About This Manual

This manual contains information to help you understand the concepts of Altibase C Interface and use Altibase C Interface.

Target Users

This manual is for Altibase users as follows.

- Database Administrators
- Performance Managers
- Database Users
- Application Program Developers
- Technical Assistance Team

This manual assumes that you have the following background :

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides.
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming
- Some experience with database server administration, operating system administartion or network administrationin

Software Dependencies

This manual assumes that your database server is Altibase server, version 5.5.1.

How This Manual is Structured

This manual covers the following topics :

- Chapter1: Introduction to Altibase C Interface

  This chapter presents an introduction to, and overview of, Altibase C Interface such as what Altibase C Interface is.

- Chapter2: Data Types

  This chapter discusses data types for ACI.

- Chapter3: Function Descriptions

  This chapter covers the specifications of Altibase C Interface functions.

- Chapter4: Prepared Statement Function Descriptions

  This chapter includes the specifications of ACI prepared statemenf functions.

- Chapter5: Using Array Binding and Array Fetching

  This chapter shows you how to use array binding and array fetching.

- Chapter6: Using Failover

  This chapter presents an introduction to functions to use failover provided with Altibase, and an overview of using them.

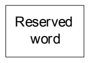
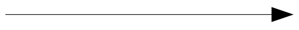
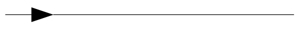
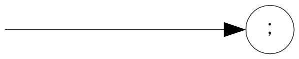
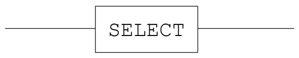
Documentation Conventions

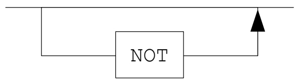
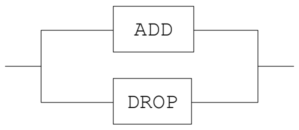
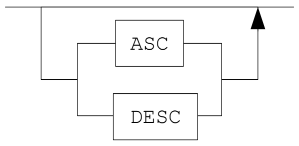
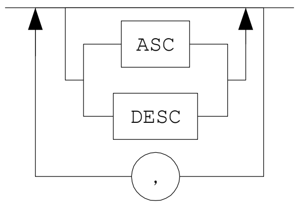
This seciton offers documentation conventions as follows. They make it easier to gather information from Altibase manuals.

- Command-Line Conventions
- Typographical 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.
	Indicates a mandatory element.

Element	Description
	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]`
{ }	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 ]`

Convention	Meaning	Example

.
.
.
	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, consult the following manuals.

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

On-Line Manual

Manuals (PDF and HTML) in Korean and English are available at Altibase Technical Center (http://atc.altibase.com/).

Altibase Welcomes Your Comments

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

Include the following information :

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

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

When you need an immediate assistance regarding technical issues, please contact Altibase Technical Center.

Thank you. We appreciate your feedback and suggestions.

1 Introduction to Altibase C Interface

This chapter describes Altibase C Interface (ACI) concepts such as what it is, and also lists features which Altibase C Interface provides.

1.1 What is Altibase C Interface?

1.1.1 Concepts

Altibase C Interface (ACI) is a mechanism for interacting with a computer operating system or software to perform specific tasks. ACI is developed in order to facilitate easier communication between applications and databases. More specifically, ACI makes possible for applications to access data from a variety of database management systems, and also provides calling level interfaces to access database servers and to execute SQL statements.

The designer of ACI aims to make it independent of programming languages, database systems, and operating systems. Thus, any application can use ACI to query data from a database, regardless of the platform it is on or database it uses.

1.1.2 ACI versus CLI

ACI contrasts with the use of CLI to type commands. CLI is used to perform various functions. However, to accomplish a successful execution, ACI need to make fewer function calls with using fewer arguments than CLI need to. You can customize ACI to meet your requirement more simply.

1.2 Using ACI

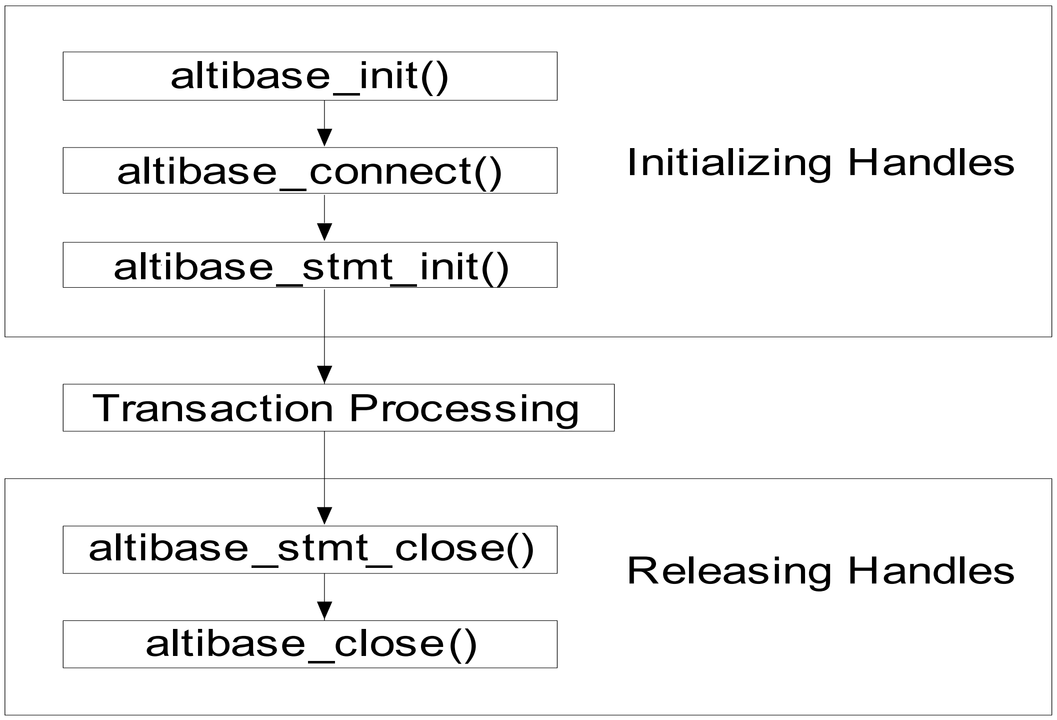
This chapter explains how to develop user application programs using ACI.

1.2.1 Basic Usages

ACI application program generally consists of following three parts:

- Initializing Handles
- Transaction Processing
- Release Handles

Besides the above, the diagnosis messages is made through the all parts of an application.



1.2.2 Initializing Handles

This is a part to allocate and initialize the environment and connection handles. Transition from one phase to the next phase is made through transmission of proper handles to send information about the execution results from the previous phase. Handle types provided by ACI are as follows:

1.2.2.1 Altibase Handle

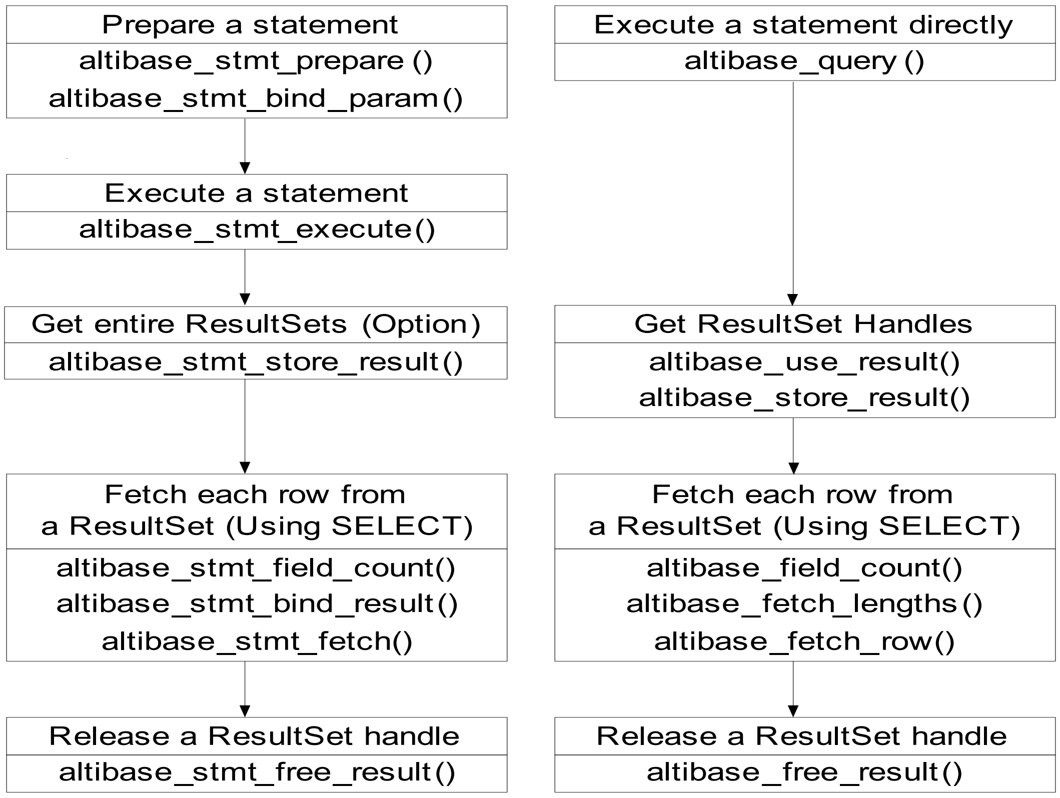
Altibase handle obtains connection-related information which ACI manages. They include connection and transaction status. An application creates and allocates Altibase handle for each connection, and then attempts to connect to Altibase. Consequently, an application can query data from Altibase.

1.2.2.2 ALTIBASE_STMT Handle

You can execute a PreparedStatement by using ALTIBASE_STMT handle. One Altibase handle can create up to 1024 ALTIBASE_STMT handles. If you want to supply values to be used in place of the question mark placeholders before you can execute a PreparedStatement, you must use ALTIBASE_STMT handle.

1.2.3 Processing of Transactions

The following figure is a general procedure of calling functions to processing a transaction.


Prepare a statement
altibase_stmt_prepare ()
altibase_stmt_bind_param()
Execute a statement
altibase_stmt_execute()
Get entire ResultSets (Option)
altibase_stmt_store_result()
Fetch each row from a ResultSet (Using SELECT)
altibase_stmt_field_count()
altibase_stmt_bind_result()
altibase_stmt_fetch()
Release a ResultSet handle
altibase_stmt_free_result()
Execute a statement directly
altibase_query ()
Get ResultSet Handles
altibase_use_result()
altibase_store_result()
Fetch each row from a ResultSet (Using SELECT)
altibase_field_count()
altibase_fetch_lengths()
altibase_fetch_row()
Release a ResultSet handle
altibase_free_result()

1.2.4 Releasing Handle

This step is for releasing the handles and meory allocated by an application, and finishing an application.

1.2.5 Managing Diagnosis Messages

Diagnosis is to handle the warning or error status occurred in an application. When calling function, you recieve return value and then can know whether function works successfully or not. For details of return value for each function, see Chapter3: Function Descriptions.

If function fails to work successfully, diagnosis messages are usually created. If you want to get more information from them in detail, you can do by using other functions as follows. Following functions are grouped depending on what handle to be used for calling functions previously.

Handle Type	Altibase	ALTIBASE_STMT	Description
Function	altibase_errno()	altibase_stmt_errno()	Error code
	altibase_error()	altibase_stmt_error()	Error message
	altibase_sqlstate()	altibase_stmt_sqlstate()	SQLSTATE message

The diagnosis messages are returned except the case of SQL_SUCCESS, SQL_NO_DATA_FOUND, SQL_INVALID_HANDLE. To check the diagnosis message, call SQLGetDiagRec(), SQLGetDiagField()

1.2.5.1 Diagnosis Messages

The diagnosis message is a five-bytes alphanumeric character string. The heading two characters refer to the class, and the next three character refer to the sub class. ACI diagnosis messages follow the standard of X/Open SQL CAE specifications.

1.2.6 Restriction

Altibase client library doesn't use signal processor. Therefore, if access to network terminates due to external factors, application can be shut down compulsorily by receiving signal of SIGPIPE. You may process it in user application to avoid forced shutdown. And you can't call functions of Altibase client library to process it because program can be stopped. However, you can after processing it.

2 Data Types

This chapter discusses data types used for Altibase C Interface. It takes a while to learn how to use them efficiently.

2.1 ACI Data Types

We will show how you can implement data types used for Altibase C Interface and provide some examples of their use. ACI provides the following categories of data types:

- Altibase handles
- Data structures
- Other data types
- Relationships among Data Types

2.1.1 Altibase Handles

This section includes the following topics:

- ALTIBASE
- ALTIBASE_RES
- ALTIBASE_STMT

2.1.1.1 ALTIBASE

This data type represents a handle to one database connection. It is used for almost all Altibase functions. You must call altibase_init() to initialize a connection handle and altibase_close() to close the connection. A connection handle can allocate only an ALTIBASE_RES. You must free a current ALTIBASE_RES to obtain new one.

2.1.1.2 ALTIBASE_RES

This data type represents the result of a query that returns rows. The information returned from a query is called the result set in the remainder of this chapter. The result set can be used to process total number of columns and individual column information.

You must call altibase_use_result() or altibase_store_result() for every statement that successfully produces a result set. You must also call altibase_free_result() after you are done with the result set.

2.1.1.3 ALTIBASE_STMT

This data type is a handle for a prepared statement. If a statement contains parameter markers or you want to get data by using the bind operation, you must use the prepared statement. You should initialize the handle with altibase_stmt_init() and close it with altibase_stmt_close().

2.1.2 Data Structures

This section includes the following topics:

- struct ALTIBASE_BIND

- struct ALTIBASE_CHARSET_INFO
- struct ALTIBASE_FIELD
- struct ALTIBASE_NUMERIC
- struct ALTIBASE_TIMESTAMP

2.1.2.1 struct ALTIBASE_BIND

The struct ALTIBASE_BIND is used to define information for the binding operation. This data type contains the following members for use by application programs.

Member	Member Type	Description
buffer_type	ALTIBASE_BIND_TYPE	This indicates the data type. For more details, see enum ALTIBASE_BIND_TYPE.
buffer	void *	This indicates a pointer to be used for data transfer. For input, buffer is a pointer to the variable in which you store the data value for a statement parameter. For output, buffer is a pointer to the variable in which to return a result set column value.
buffer_length	ALTIBASE_LONG	This indicates the actual size of buffer. You do not have to define data types whose lengths are fixed across platforms as follows for the binding operation. To achieve this, their lengths must be set to 0 after initialization. ALTIBASE_BIND_SMALLINT, ALTIBASE_BIND_INTEGER, ALTIBASE_BIND_BIGINT, ALTIBASE_BIND_REAL, ALTIBASE_BIND_DOUBLE, ALTIBASE_BIND_DATE You must set buffer_length to a valid value when binding a string variable by specifying a data type such as ALTIBASE_BIND_STRING whose length is not fixed. If the size of actual data is greater than a value of buffer_length, data can be buffered only as much as you set a value. For example, if you specify buffer_length as 2, 2bytes from the starting are buff-ered in the value of int. You must set buffer_length to a valid value to return a valid result.

Member	Member Type	Description
length	ALTIBASE_LONG *	This indicates the actual number of bytes of data. You do not have to define data types such as short and int whose lengths are variable across platforms. You must define character string or binary data as a valid value because the sizes of character string and binary data may be smaller than that of buffer. You can use ALTIBASE_NULL_DATA to fetch data. This indicates the return value is null.
is_null	ALTIBASE_BOOL *	This member points to an ALTIBASE_BOOL* variable that is ALTIBASE_TRUE if a value is null. It is recommended to check if a value is null by using this variable before using a value.
error	int	This member points to an int variable to have information for the parameter stored after the binding operation. When the binding operation fails, you can check what argument fails specifically by using this variable. A value is returned by calling altibase_errno(). For more details, see 3.15 altibase_errno().

altibase_stmt_bind_param() and altibase_stmt_bind_result() are used to set the bind argument.

2.1.2.2 struct ALTIBASE_CHARSET_INFO

The struct ALTIBASE_CHARSET_INFO is used to define information for charater set.

Member	Member Type	Description
id	unsigned int	The identification of character set
name	void *	The name of character set encoded as UTF8
name_length	int	The name length of chaset
mbmaxlen	int	The maximum length of one character (Unit : Byte)

You may obtain members for each field by calling altibase_get_charset_info().

2.1.2.3 struct ALTIBASE_FIELD

This structure contains information about a field. The struct ALTIBASE_FIELD contains the members described in the following list.

Member	Member Type	Description
type	ALTIBASE_FIELD_TYPE	The type of the field
name	char [ALTIBASE_MAX_FIELD_NAME_LEN]	The name of the field
name_length	int	The length of field name
org_name	char [ALTIBASE_MAX_FIELD_NAME_LEN]	The name of the original field
org_name_length	int	The length of org_name
table	char [ALTIBASE_MAX_TABLE_NAME_LEN]	The name of the table containing this field
table_length	int	The length of table name
org_table	char [ALTIBASE_MAX_TABLE_NAME_LEN]	The name of the original table
org_table_length	int	The length of org_table
size	int	The size or precision of the field
scale	int	The numerical scale

You may obtain members for each field by calling altibase_field() or altibase_fields(). You must not change or release them as you please because they are managed within procedure.

2.1.2.4 struct ALTIBASE_NUMERIC

The struct ALTIBASE_NUMERIC is used to send and receive numerical data to and from the server.

Member	Member Type
precision	unsigned char
scale	unsigned char
sign	char
val	unsigned char [ALTIBASE_MAX_NUMERIC_LEN]

2.1.2.5 struct ALTIBASE_TIMESTAMP

The struct ALTIBASE_TIMESTAMP is used to send and receive date data to and from the server.

Member	Member Type	Description
year	short	The year
month	unsigned short	The month of the year
day	unsigned short	The day of the month
hour	unsigned short	The hour of the day
minute	unsigned short	The minute of the hour
second	unsigned short	The second of the minute
fraction	int	One over one hundred thousand second

2.1.3 Other Data Types

This section includes the following topics:

- ALTIBASE_ROW
- ALTIBASE_LONG
- ALTIBASE_NTS
- enum ALTIBASE_BIND_TYPE
- enum ALTIBASE_FAILOVER_EVENT
- enum ALTIBASE_FIELD_TYPE
- enum ALTIBASE_OPTION
- enum ALTIBASE_STMT_ATTR_TYPE

2.1.3.1 ALTIBASE_ROW

This is a type-safe representation of one row of data. Rows are obtained by calling altibase_fetch_row() when altibase_query() is used with a statement such as SELECT which returns a reuslt set.

A field value contains binary data or character string. If fields have data types such as BLOB, BYTE, NIBBLE, BIT, VARBIT or GEOMETRY, fields are encoded as binary data. Otherwise, fields are encoded as character string.

The NIBBLE, BIT and VARBIT are all based on binary logic in special form. A NIBBLE is a four-bit aggregation and a BIT is the basic unit of information. To obtain these easily, database effectively performs the required macro substitutions by using GET_NIBBLE_VALUE() and GET_BIT_VALUE(). You must not change or release their values as you please because they are managed within procedure.

2.1.3.2 ALTIBASE_LONG

This can be defined as a 32-bit integer or a 64-bit integer. This works similarly to SQLLEN defined by the Altibase ODBC driver. This is used to get row number or the number of rows.

2.1.3.3 ALTIBASE_NTS

This represents the null-terminated string. If you want to specify the length, ALTIBASE_NTS can be used instead of actual length. If finding the actual length of a string, you can use the strlen() function to do the job for you. You must not specify binary data as ALTIBASE_NTS because strlen() returns wrong value.

2.1.3.4 enum ALTIBASE_BIND_TYPE

This gets the data type of the bind variable as follows.

enum Value	Data Type
ALTIBASE_BIND_NULL	This is used to input null only for the parameter binding. This setting is similar to specifying that is_null which is one of members in ALTIBASE_BIND points to an ALTIBASE_BOOL* variable that is ALTIBASE_TRUE.
ALTIBASE_BIND_BINARY	This is used for binary data whose type is BYTE, NIBBLE, BIT, VARBIT, BLOB or GEOMETRY.
ALTIBASE_BIND_STRING	This is used for character strings such as CHAR, VARCHAR, NCHAR and NVARCHAR.
ALTIBASE_BIND_WSTRING	This is used for unicode character.
ALTIBASE_BIND_SMALLINT	This is used for the SMALLINT type which is a 16 bit sized signed integer.
ALTIBASE_BIND_INTEGER	This is used for the INTEGER type which is a 32 bit sized signed integer.
ALTIBASE_BIND_BIGINT	This is used for the BIGINT type which is a 64 bit sized signed integer.
ALTIBASE_BIND_REAL	This is used for the REAL type which is a single prcision floating-point number.
ALTIBASE_BIND_DOUBLE	This is used to represent the DOUBLE type which is a double pre-floating-point number.
ALTIBASE_BIND_NUMERIC	This is used to store numeric data types such as NUMERIC, DECIMAL NUMBER and FLOAT.
ALTIBASE_BIND_DATE	This is used to represent DATE type storing date and time values.

2.1.3.5 enum ALTIBASE_FAILOVER_EVENT

This gets the information for the failover events as follows.

enum Value	Data Type
ALTIBASE_FO_BEGIN	This notifies the start of STF (Service Time Failover).
ALTIBASE_FO_END	This notifies the success of STF.
ALTIBASE_FO_ABORT	This notifies the failure of STF.
ALTIBASE_FO_GO	FailOverCallback sends this so that STF can advance to the next step.
ALTIBASE_FO_QUIT	FailOverCallback sends this to prevent STF from advancing to the next step.

If you register the failover callback function, the failover callback function is notified of values returned by the failover events. They are used when the failover callback function determines its advance to the next step. For more details, see Chapter6: Using Failover.

2.1.3.6 enum ALTIBASE_FIELD_TYPE

This gets the data type of the specified field as follows.

enum Value	Data Type
ALTIBASE_TYPE_CHAR	CHAR
ALTIBASE_TYPE_VARCHAR	VARCHAR
ALTIBASE_TYPE_NCHAR	NCHAR
ALTIBASE_TYPE_NVARCHAR	NVARCHAR
ALTIBASE_TYPE_SMALLINT	SMALLINT
ALTIBASE_TYPE_INTEGER	INTEGER
ALTIBASE_TYPE_BIGINT	BIGINT
ALTIBASE_TYPE_REAL	REAL
ALTIBASE_TYPE_DOUBLE	DOUBLE
ALTIBASE_TYPE_FLOAT	FLOAT
ALTIBASE_TYPE_NUMERIC	NUMERIC
ALTIBASE_TYPE_DATE	DATE
ALTIBASE_TYPE_BLOB	BLOB
ALTIBASE_TYPE_CLOB	CLOB

enum Value	Data Type
ALTIBASE_TYPE_BYTE	BYTE
ALTIBASE_TYPE_NIBBLE	NIBBLE
ALTIBASE_TYPE_BIT	BIT
ALTIBASE_TYPE_VARBIT	VARBIT
ALTIBASE_TYPE_GEOMETRY	GEOMETRY

You can use IS_NUM_TYPE() to check numeric data types such as SMALLINT, INTEGER, BIGINT, REAL, DOUBLE, FLOAT and NUMERIC. You can use IS_BIN_TYPE() to check binary string types such as BYTE, BLOB, NIBBLE, BIT, VARBIT and GEOMETRY. You cannot check if binary byte strings are stored in the column specified as CHAR.

2.1.3.7 enum ALTIBASE_OPTION

This gets the information for the connect options as follows.

enum Value	Data Type	Maximum Size	Description
ALTIBASE_AUTOCOMMIT	int	sizeof(int)	This is used to set connect options and affect behavior for a connection. This represents a 32 bit. ALTIBASE_AUTOCOMMIT_ON: each individual SQL statement is automatically committed right after it is executed. ALTIBASE_AUTOCOMMIT_OFF: each individual SQL statement is not automatically committed right after it is executed.
ALTIBASE_CONNECTION_TIMEOUT	int	sizeof(int)	This is used to set the value of timeout to make a connection to the database server in a nonblocking manner. Blocking can be caused by using the select() or poll() when network is unstable.
ALTIBASE_PORT	int	sizeof(int)	This is used to define server port.
ALTIBASE_TXN_ISOLATION	int	sizeof(int)	This is used to determine the transaction isolation level for current connection.

enum Value	Data Type	Maximum Size	Description
ALTIBASE_APP_INFO	char *	ALTIBASE_MAX_APP_INFO_LEN	The client uses the application ID retrieved using ALTIBASE_APP_INFO. You can know speinformation of user's session by using this because ALTIBASE_APP_INFO runs consent in your session.
ALTIBASE_DATE_FORMAT	char *	ALTIBASE_MAX_DATE_FORMAT_LEN	This is used to display date data in formats. The default is YYYY/MM/DD HH:MI:SS.
ALTIBASE_NLS_USE	char *	ALTIBASE_MAX_NLS_USE_LEN	This is used to select language. (US7ASCII : English character set, KO16KSC5601 : Korean chaset)
ALTIBASE_NLS_NCHAR_LITERAL_REPLACE	int	sizeof(int)	This determines whether to check national character set data by parsing a SQL statement.
ALTIBASE_IPC_FILEPATH	chat *	ALTIBASE_MAX_IPC_FILEPATH_LEN	The UNIX domain can be used to communicate between processes on IPC. One process usually acts as a server and the other process is the client. The UNIX domain provides a socket address space on IPC. Communicating processes connect through addresses In the UNIX domain, a connection is usually composed of one path name as ALTIBASE_HOME. The server binds its socket to a previously agreed path or address. However, if two processes connect through different paths of ALTIBASE_HOME respectively, you cannot establish a connection between them because a socket domain also provides different addressing structures. At this time, if you use ALTIBASE_HOME/trc/cm-ipc file, the Unix domain is available. Therefore, you can pass data retrieved from a shared memory between processes.

It is recommended to use altibase_set_autocommit() when you want to set ALTIBASE_AUTOCOMMIT.

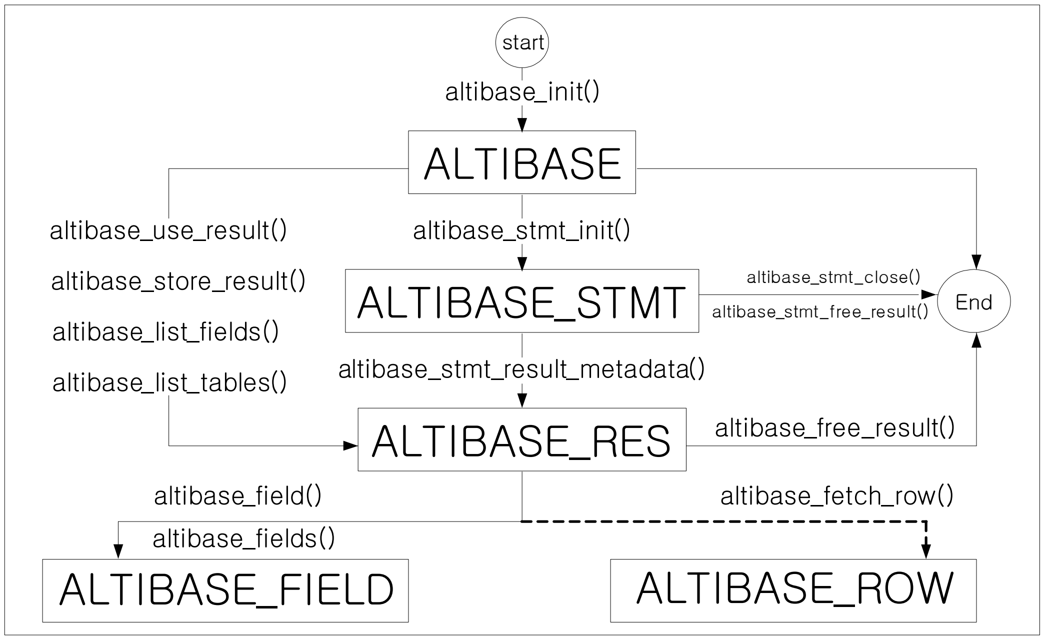
2.1.3.8 enum ALTIBASE_STMT_ATTR_TYPE

This gets the information for attribute types of statement handle as follows.

enum Value	Data Type	Maximum Size	Description
ALTIBASE_STMT_ATTR_ATOMIC_ARRAY	int	sizeof(int)ar	This specifies whether all of the rows should be inserted as an atomic operation or not. ARRAY INSERT inserts a single data into a given array separately and independently. In comparison, the multiple inserts are processed as a single statement when you use the ATOMIC ARRAY INSERT. You can set ALTIBASE_ATOMIC_ARRAY_ON or ALTIBASE_ATOMIC_ARRAY_OFF. If other value is set except them, error occurs during the execution of an insert. If you set ALTIBASE_ATOMIC_ARRAY_ON, ATOMIC ARRAY INSERT is in effect. Array size effects in performance. Therefore, you should declare right size of array. You would realize the performance improvements with using ATOMI ARRAY INSERT rather than ARRAY INSERT. When a_stmt_status() and altibase_stmt_processed() returns the result, only the first value of the result is valid because multiple inserts are processed as a single statement by using the ATOMIC ARRAY INSERT.

2.1.4 Relationships among Data Types

The following figure illustrates the relationships among Altibase handles and other data types.



You may obtain ALTIBASE_RES by using ALTIBASE_STMT. However, you may not obtain ALTIBASE_ROW returned by ALTIBASE_RES. The dotted line in figure means this relationship between ALTIBASE_ROW and ALTIBASE_RES.

You may obtain ALTIBASE_ROW only by using altibase_fetch_row() related to altibase_query() executing the SQL statement without the binding operaiton. altibase_fetch_row() retrieves the next row of data from a result handle returned by altibase_query().

For more detail, see Chapter3: Function Descriptions.

3 Function Descriptions

This chapter describes the specifications of ACI functions used with Altibase handle. For each ACI functions, the following information are described.

- Name of the function and purpose of use
- Arguments list of the function
- Return Values
- Usages of function and notes
- Diagnosis message that can be displayed when an error occurrs in function
- Example source codes

See *Error Message Reference* for error messages related to using these functions.

3.1 altibase_affected_rows()

altibase_affected_rows() may be called immediately after executing a statement. It returns the number of rows changed, deleted, or inserted by the last statement if it was an UPDATE, DELETE, or INSERT.

3.1.1 Syntax

```
ALTIBASE_LONG altibase_affected_rows
(
    ALTIBASE altibase
)
```

3.1.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection Handle

3.1.3 Return Values

Return Value	Description
Greater than 0	An integer indicates the number of rows affected or retrieved.
0	Zero indicates that no rows were updated or that no query has yet been executed.
ALTIBASE_INVALID_AFFECTEDROWS	This indicates that the query returned an error.

3.1.4 Description

altibase_affected_rows() returns the value that it would return for the last statement executed within the procedure. For UPDATE statements, the affected-rows value by default is the number of rows actually changed. For DELETE statements, the affected-rows value is the number of deleted rows. For INSERT statements, the affected-rows value is the number of existing rows which are updated. For SELECT statements, the affected-rows value is 0, and altibase_affected_rows() works like altibase_num_rows() which returns the number of rows selected by a SELECT statement.

3.1.5 Example

```
#define QSTR "UPDATE employees SET salary = salary * 1.1 WHERE group = 1"

rc = altibase_query(altibase, QSTR);
/* ... check return value ... */

printf("%ld updated\n", altibase_affected_rows(altibase));
```

3.2 altibase_client_version()

altibase_client_version() returns a constant that represents the client library version.

3.2.1 Syntax

```
int altibase_client_version (void)
```

3.2.2 Return Values

altibase_client_version() returns the client library version in a numeric format.

3.2.3 Description

altibase_client_version() returns the client library information as a constant. The value has the format `Mmmddpppp` whose specific meaning is as follows.

Format	Meaning	Note
M	Where `M` is the major version.	
mm	Where `mm` is the minor version.	When you assign a value to `mm`, if the value is shorter than the declared length of this, Altibase pads 0 to the rest space.
dd	Where `dd` is the development version.	When you assign a value to `dd`, if the value is shorter than the declared length of this, Altibase pads 0 to the rest space.
pppp	Where `pppp` is the patch level.	When you assign a value to `pppp`, if the value is shorter than the declared length of this, Altibase pads 0 to the rest space.

For example, a value of 503050001 represents a client library version of 5.3.5.1.

3.3 altibase_client_verstr()

altibase_client_verstr() returns a string that represents the client library version.

3.3.1 Syntax

```
const char * altibase_client_verstr (void)
```

3.3.2 Return Values

altibase_client_verstr() returns a string that represents the client library version.

3.3.3 Description

altibase_client_verstr() returns a string that represents the client library version. The value has the format x.x.x.x. You must not change or release it as you please because it is manged within procedure.

3.4 altibase_close()

altibase_close() closes a previously opened connection.

3.4.1 Syntax

```
int altibase_close
(
    ALTIBASE altibase
)
```

3.4.2 Arguments

Data Type	Arguments	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.4.3 Return Values

altibase_close() returns ALTIBASE_SUCCESS on success or ALTIBASE_ERROR on failure.

3.4.4 Description

altibase_close() closes an opened connection, and removes entire resources allocated to connection handle. If you free Altibase handle, ALTIBASE_STMT and ALTIBASE_RES handles are automatically invalid. Therefore, you must finish task required to use ALTIBASE_STMT or ATLIBASE_RES handle, and then free ALTIBASE_STMT or ATLIBASE_RES handle which is subordinate to ATLIBASE handle before freeing Altibase handle.

3.4.5 Example

```
altibase = altibase_init();
if (altibase == NULL)
{
    return 1;
}

/* ... omit ... */

rc = altibase_close(altibase);
/* ... check return value ... */
```

3.5 altibase_commit()

altibase_commit() commits the current trasnaction. In other words, the function commits changes made to connection. After this function is executed, statements are written to the database.

3.5.1 Syntax

```
int altibase_commit
(
    ALTIBASE altibase
)
```

3.5.2 Arguments

Data Type	Arguments	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.5.3 Return Values

The function returns ALTIBASE_SUCCESS if successful or ALTIBASE_ERROR if unsuccessful.

3.5.4 Description

altibase_commit() commits changes made to your session related to connection. If you have the autocommit mode off, new transaction is created internally when you issue a statement which can be included in a transaction.

3.5.5 Example

See the example in altibase_set_autocommit().

3.6 altibase_rollback()

altibase_rollback() rolls back the current transaction. In other words, the function aborts the queries you have sent before and reverts them to the old values in the database.

3.6.1 Syntax

```
int altibase_rollback
(
    ALTIBASE altibase
)
```

3.6.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection Handle

3.6.3 Return Values

The function returns ALTIBASE_SUCCESS if successful or ALTIBASE_ERROR if an error occurred.

3.6.4 Description

altibase_rollback() rolls back the current transaction for your session related to connection. If you have the autocommit mode off, new transaction is created internally when you issue a statement which can be included in a transaction.

3.6.5 Example

See the example in altibase_set_autocommit().

3.7 altibase_set_autocommit()

altibase_set_autocommit() sets the autocommit mode to on.

3.7.1 Syntax

```
int altibase_set_autocommit
(
    ALTIBASE altibase,
    int      mode
)
```

3.7.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle
int	mode	Input	This determines whether the autocommit mode is set to on.

3.7.3 Return Values

The function returns ALTIBASE_SUCCESS if successful or ALTIBASE_ERROR if an error occurred.

3.7.4 Description

The value of the autocommit mode can be ALTIBASE_AUTOCOMMIT_ON or ALTIBASE_AUTOCOMMIT_OFF. If it cannot, error occurs. altibase_set_autocommit() enables the autocommit mode if ALTIBASE_AUTOCOMMIT_ON is set, or disables it if ALTIBASE_AUTOCOMMIT_OFF is set. By default, autocommit is enabled.

3.7.5 Example

```
rc = altibase_set_autocommit(altibase, ALTIBASE_AUTOCOMMIT_OFF);
/* ... check return value ... */

/* ... omit ... */

rc = (error_exist) ? altibase_rollback(altibase) : altibase_commit(altibase);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    /* ... error handling ... */
}

rc = altibase_set_autocommit(altibase, ALTIBASE_AUTOCOMMIT_ON);
/* ... check return value ... */
```

3.8 altibase_connect()

altibase_connect() attempts to establish a connection to an Altibase database engine running on host by using connection string.

3.8.1 Syntax

```
int altibase_connect
(
    ALTIBASE    altibase,
    const char *connstr
)
```

3.8.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle
const char *	connstr	Input	Coneection string

3.8.3 Return Values

The function returns ALTIBASE_SUCCESS if successful or ALTIBASE_ERROR if an error occurred.

3.8.4 Description

Connection string is used to send each value of more than one parameter such as DSN, PORT_NO, UID, PWD, CONNTYPE or NLS_USE. The parameters are required to establish a connection. For more details, see *ODBC Reference*. Connection string must be a terminating null character.

3.8.4.1 Example

```
#define CONNSTR "DSN=127.0.0.1;PORT_NO=20300;UID=sys;PWD=manager"

ALTIBASE altibase;

altibase = altibase_init();
/* ... check return value ... */

rc = altibase_set_option(altibase, ALTIBASE_APP_INFO, "your_app_name");
/* ... check return value ... */

rc = altibase_connect(altibase, CONNSTR);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    fprintf(stderr, "Failed to connect : %s\n", altibase_error(altibase));
}
```

3.9 altibase_data_seek()

altibase_data_seek() seeks to an arbitrary row in a query result set and changes the pointer location of the resource.

3.9.1 Syntax

```
int altibase_data_seek
(
    ALTIBASE_RES  result,
    ALTIBASE_LONG offset
)
```

3.9.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_RES	result	Input	Result handle
ALTIBASE_LONG	offset	Input	The offset value is a row number and should be in the range from 0.

3.9.3 Return Values

The function returns ALTIBASE_SUCCESS if successful or ALTIBASE_ERROR if an error occurred.

3.9.4 Description

altibase_data_seek() moves the internal row pointer of the result associated with the specified result indentifier to point to the specified row number. The offset value is a row number and should be in the range from 0 to altibase_num_rows(result) - 1. altibase_data_seek() may be used only in conjunction with altibase_store_result().

3.9.5 Example

```
#define QSTR "SELECT last_name, first_name FROM friends"

rc = altibase_qeury(altibase, QSTR);
/* ... check return value ... */

result = altibase_store_result(altibase);
/* ... check return value ... */

row_count = altibase_num_rows(result);
for (i = 0; i < row_count; i++)
{
    rc = altibase_data_seek(result, i);
```

```
        if (ALTIBASE_NOT_SUCCEEDED(rc))
        {
            printf("ERR : %d : ", i, altibase_error());
            continue;
        }

        /* ... omit ... */
    }

    rc = altibase_free_result(result);
    /* ... check return value ... */
```

3.10 altibase_fetch_row()

altibase_fetch_row() retrieves a row of a result set.

3.10.1 Syntax

```
ALTIBASE_ROW altibase_fetch_row
(
    ALTIBASE_RES result
)
```

3.10.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_RES	result	Input	Result handle

3.10.3 Return Values

altibase_fetch_row() returns data in a row on success, or null if error occurs or no rows are left.

3.10.4 Description

altibase_fetch_row() fetches one row of data from the result set. The function returns null if error occurs or no rows are left. When used after altibase_store_result(), altibase_fetch_row() returns null when there are no more rows to retrieve.

A numerical array corresponds to a fetched row. The offset value is a column number and should be in the range from 0 to altibase_num_fields(result) - 1.

The value can contain a character string or binary data because this is type-safe representation of one row of data. If you want to treat a value as a number, you must convert the string yourself. For more details, see 2.1.3.1 ALTIBASE_ROW.

If a value has null, null values are represented by null pointers in the ALTIBASE_ROW array. The lengths of the values in the row may be obtained by calling altibase_fetch_lengths(). You must not get string length by calling strlen() because their lengths returned by altibase_fetch_row() can contain binary data.

You must use the lengths of the values by calling altibase_fetch_lengths(). altibase_fetch_row() returns data storing the lengths of each result column in a row. Therefore, there can be insufficient memory if result set contains large amounts of data such as LOB or geometry. In case like that, it is more convenient to use a parepared statement for sending SQL statements to the database with separating data. Prepared statements are designed in a more secure and efficient manner. If you want to execute a statement many times, it normally reduces execution time to use a prepared statement instead.

The value returned by altibase_fetch_row() is valid only before calling altibase_fetch_row() again. You must store the value in the row variable of application to remember. You must not change or release it as you please because it is manged within procedure.

3.10.5 Example

See the example in altibase_query().

3.11 altibase_free_result()

altibase_free_result() frees the memory allocated for a result set.

3.11.1 Syntax

```
int altibase_free_result
(
    ALTIBASE_RES result
)
```

3.11.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_RES	altibase	Input	Result handle

3.11.3 Return Values

The function returns ALTIBASE_SUCCESS if successful or ALTIBASE_ERROR if unsuccessful.

3.11.4 Description

altibase_free_result() frees all memory associated with the result set. When done with a result set returned by the following functions, you must free the memory it uses by calling altibase_free_result().

- altibase_store_result()
- altibase_use_result()
- altibase_list_fields()
- altibase_list_tables()

After this, you cannot call them required to use result set.

3.11.5 Example

See the example in altibase_query().

3.12 altibase_query()

altibase_query() executes the SQL statement.

3.12.1 Syntax

```
int altibase_query
(
    ALTIBASE    altibase,
    const char *qstr
)
```

3.12.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle
const char *	qstr	Input	The SQL statement pointed to by the null-terminated string.

3.12.3 Return Values

altibase_query() returns ALTIBASE_SUCCESS if the statement was successful. The function returns ALTIBASE_ERROR if an error occurred.

3.12.4 Description

If altibase_query() sends a query to Altibase, the SQL statement must be pointed by the null-terminated string which should consist of a single SQL statement. Multiple-statement execution has not been enabled. The string cannot contain several statements separated by semicolons. Enabling multiple-statement execution with this function need to permit processing of stored procedures.

3.12.5 Example

```
#define QSTR "SELECT last_name, first_name FROM friends"

ALTIBASE       altibase;
ALTIBASE_RES   result;
ALTIBASE_ROW   row;
ALTIBASE_LONG *lengths;
int            num_fields;
int            rc;
int            i;

/* ... omit ... */
```

```
rc = altibase_qeury(altibase, QSTR);
/* ... check return value ... */

result = altibase_use_result(altibase);
/* ... check return value ... */

num_fields = altibase_num_fields(result);
while ((row = altibase_fetch_row(result)) != NULL)
{
   lengths = altibase_fetch_lengths(result);
   for (i = 0; i < num_fields; i++)
   {
       printf("(%ld) %s", lengths[i], (row[i] == NULL ? "null" : row[i]));
   }
   printf("\n");
}

rc = altibase_free_result(result);
/* ... check return value ... */

/* ... omit ... */
```

3.13 altibase_store_result()

The statement can produce a result set successfully by calling altibase_store_result().

3.13.1 Syntax

```
ALTIBASE_RES altibase_store_result
(
    ALTIBASE altibase
)
```

3.13.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.13.3 Return Values

altibase_store_result() returns an ALTIBASE_RES result structure with the results for success, or null if an error occurred.

3.13.4 Description

altibase_store_result() returns the contents of one cell from an Altibase result set of a query. If you call altibase_store_result(), the function reads the entire result of a query to the client and allocates a ALTIBASE_RES structure. And then the function places the result into this structure. It is not neccessary to communicate with the client when you call () because entire result is already stored. Therefore, altibase_fetch_row() returns the values which are already placed by altibase_store_result().

Great attention should be paid to call altibase_store_result() because there can be insufficient memory if result set contaions large amounts of data such as LOB or geometry. An empty result set is returned instead of null if there are no row returned. Therefore, if you have called altibase_store_result() and gotten back a result that is a null pointer for a SELECT statement, you can know error occurs. If calling altibase_store_result() instead of altibase_use_result(), you can use the followings additionally.

- altibase_num_rows()
- altibase_data_seek()

altibase_store_result() cannot be used with the functions such as altibase_use_result() and altibase_list_tables() which return result set. You must call altibase_free_result() to free current result set handle and obtain other one after you are done with the result set.

3.13.5 Example

See the examples in altibase_data_seek() and altibase_query().

3.14 altibase_use_result()

You can get result set by calling altibase_use_result().

3.14.1 Syntax

```
ALTIBASE_RES altibase_use_result
(
   ALTIBASE altibase
)
```

3.14.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.14.3 Return Values

altibase_use_result() returns an ALTIBASE_RES result structure for success, or null if an error occurred.

3.14.4 Description

altibase_use_result() returns result set of a query. This function does not actually read the result set into the client like altibase_store_result() does. Instead, each row must be retrieved individually by making calls to altibase_fetch_row(). This reads the result of a query directly from the server without storing it in a temporary table or local buffer.

An empty result set is returned instead of null if there is no row returned. Therefore, if you have called altibase_use_result() and gotten back a result that is a null pointer for a SELECT statement, you can know error occurs.

altibase_use_result() cannot be used with the functions such as altibase_store_result() and altibase_list_tables() which return result set. You must call altibase_free_result() to free current result set handle and obtain other one after you are done with the result set.

3.14.5 Example

See the example in altibase_query().

3.15 altibase_errno()

For the connection specified by Altibase, altibase_errno() returns the error code for the most recently invoked API function that can succeed or fail.

3.15.1 Syntax

```
unsigned int altibase_errno
(
    ALTIBASE altibase
)
```

3.15.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.15.3 Return Values

0 means no error occurred. An error code value for the last altibase_errno() call is returned if it failed.

3.15.4 Description

altibase_errno() returns the numerical value of the error code from previous function. All functuins do not return error codes. Error codes are returned by queries for their operation. Errors are listed at *Error Message Refrence* in detail.

Make sure you check the value before calling another function because it is initialized or new one is created instead if you call another function. The value returned by altibase_errno() is different from that of SQLSTATE. You should use altibase_sqlstate() to find a specific SQLSTATE when handling errors.

3.15.5 Example

```
rc = altibase_query(altibase, QSTR);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    printf("error no  : %05X\n", altibase_errno(altibase));
    printf("error msg : %s\n", altibase_error(altibase));
    printf("sqlstate  : %s\n", altibase_sqlstate(altibase));
    return 1;
}

/* ... omit ... */
```

3.16 altibase_error()

For the connection specified by Altibase, altibase_error() returns error message for the most recently invoked API function.

3.16.1 Syntax

```
const char * altibase_error
(
    ALTIBASE altibase
)
```

3.16.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection Handle

3.16.3 Return Values

altibase_error() returns the error text from the last function, or an empty string if no error occurred.

3.16.4 Description

altibase_error() returns error message from the last function that failed. Make sure you check the value before calling another function because it is initialized or new one is created instead if you call another function. You must not change or release it as you please because it is manged within procedure.

3.16.5 Example

See the example in altibase_errno().

3.17 altibase_sqlstate()

altibase_sqlstate() returns a null-terminated string containing the SQLSTATE error code for the most recently executed SQL statement.

3.17.1 Syntax

```
const char * altibase_sqlstate
(
    ALTIBASE altibase
)
```

3.17.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.17.3 Return Values

altibase_sqlstate() returns a null-terminated string containing the SQLSTATE error code.

3.17.4 Description

altibase_sqlstate() returns a null-terminated string containing the SQLSTATE error code for the most recently executed SQL statement. The error code consists of five characters. '00000' means "no error". For a list of possible values of SQLSTATE, see *Altibase Error Message Reference*.

SQLSTATE values returned by altibase_sqlstate() differ from Altibase-specific error numbers returned by altibase_errno(). It is recommanded not to check the values returned by altibase_errno() but those of SQLSTATE if you need error code.

Not all Altibase error numbers returned by altibase_errno() are mapped to SQLSTATE error codes. Therefore, you cannot know the values of SQLSTATE by checking thoses returned by altibase_errno(), or you cannot know the values returned by altibase_errno() by checking those of SQLSTATE exactly.

Make sure you check the value before calling another function because it is initialized or new one is created instead if you call another function. You must not change or cancel it as you please because it is managed within procedure.

3.17.5 Example

See the example in altibase_errno().

3.18 altibase_fetch_lengths()

altibase_fetch_lengths() returns the lengths of the columns of the current row within a result set.

3.18.1 Syntax

```
ALTIBASE_LONG * altibase_fetch_lengths
(
    ALTIBASE_RES result
)
```

3.18.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_RES	result	Input	Result Handle

3.18.3 Return Values

altibase_fetch_lengths() returns an array of unsigned long integer representing the size of each column on success, or null if an error occurred.

3.18.4 Description

If altibase_fetch_lengths() returns an array that corresponds to the lengths of each column in the current row fetched by Altibase. In case of a character string column, the function returns its length without including any terminating null characters Therefore, you must also consider the length of a null-termination character to create buffer.

If column is specified as null, ALTIBASE_NULL_DATA is returned for legnth of column. altibase_fetch_lengths() is valid only for the current row of the result set. Therefore, you must call altibase_fetch_lengths() after calling altibase_fetch_row(). altibase_fetch_lengths() returns null if you call it before calling altibase_fetch_row() or fetch no rows which are left.

You must avoid calling strlen() to check the length because data returned by altibase_fetch_row() can contain binary data. You should call altibase_fetch_lengths() to check data length. You must not change or release it as you please because it is managed within procedure.

3.18.5 Example

```
ALTIBASE_ROW   row;
ALTIBASE_LONG *lengths;
int            num_fields;
int            i;

/* ... omit ... */
```

```
num_fields = altibase_num_fields(result);
row = altibase_fetch_row(result);
if (row != NULL)
{
    lengths = altibase_fetch_lengths(result);
    for (i = 0; i < num_fields; i++)
    {
         printf("Column length %d : %ld\n", i, lengths[i]);
    }
}

/* ... omit ... */
QLDisconnect( dbc );
```

3.19 altibase_field()

altibase_field() returns the definition of one column of a result set.

3.19.1 Syntax

```
ALTIBASE_FIELD * altibase_field
(
    ALTIBASE_RES result,
    int          fieldnr
)
```

3.19.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_RES	result	Input	Result handle
int	fieldnr	Input	This is a column number which starts at 0.

3.19.3 Return Values

altibase_field() returns the pointer to the definition of a specified column on success, or null if error occurs or no columns are left.

3.19.4 Description

altibase_field() returns the pointer of the definition of a specified column. The offset value is a column number and should be in the range from 0 to altibase_num_fields(result) - 1. You must not change or release it as you please because it is manged within procedure.

3.19.5 Example

```
ALTIBASE_FIELD *field;
int num_fields;
int i;

num_fields = altibase_num_fields(result);
for (i = 0; i < num_fields; i++)
{
    field = altibase_field(result, i);
    printf("%d : %s\n", i, field->name);
}
```

3.20 altibase_field_count()

altibase_field_count() returns the number of columns for the most recent query on the connection.

3.20.1 Syntax

```
int altibase_field_count
(
    ALTIBASE altibase
)
```

3.20.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.20.3 Return Values

Return Value	Description
Greater than 0	An integer indicates the number of columns in result set.
0	Zero indicates that no result sets are left.
ALTIBASE_INVALID_FIELDCOUNT	Error occurs.

3.20.4 Description

altibase_field_count() returns the number of columns for the most recent query. This enables the client program to take proper action with returning 0 if the query was a SELECT statement.

3.20.5 Example

```
/* ... omit ... */

rc = altibase_query(altibase, qstr);
/* ... check return value ... */

printf("field count = %d\n", altibase_field_count(altibase));
```

3.21 altibase_get_charset()

altibase_get_charset() returns character set name for the current connection.

3.21.1 Syntax

```
const char * altibase_get_charset
(
    ALTIBASE altibase
)
```

3.21.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.21.3 Return Values

altibase_get_charset() returns the name of character set derived from NLS_USE environment variable.

3.21.4 Description

altibase_get_charset() returns character set name for the current connection. It can be derived from NLS_USE environment variable or connection string, or can be defined by using altibase_set_charset() before sending data from and to the database server. If it is not set, it returns the default character set. You must not change or release it as you please because it is manged within procedure.

3.21.5 Example

```
rc = altibase_set_charset(altibase, "KO16KSC5601");
/* ... check return value ... */

printf("NLS_USE = %s\n", altibase_get_charset(altibase));
```

3.22 altibase_get_charset_info()

This function is not enabled currently.

3.23 altibase_host_info()

This function is not enabled currently.

3.24 altibase_init()

altibase_init() allocates or initializes an Altibase object as a connection handle.

3.24.1 Syntax

```
ALTIBASE altibase_init (void)
```

3.24.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection Handle

3.24.3 Return Values

altibase_init() returns an initialized Altibase connection handle on success, or null if it failed.

3.24.4 Description

altibase_init() allocates an Altibase object as a connection handle suitable for altibase_connect(). If altibase_init() allocates a new object as a connection handle, it is freed when altibase_close() is called to close the connection.

3.24.5 Example

```
altibase = altibase_init();
if (altibase == NULL)
{
    return 1;
}

/* ... omit ... */

rc = altibase_close(altibase);
/* ... check return value ... */
```

3.25 altibase_list_fields()

altibase_list_fields() returns a result set consisting of field names in the given table.

3.25.1 Syntax

```
ALTIBASE_RES altibase_list_fields
(
    ALTIBASE    altibase,
    const char *restrictions[]
)
```

3.25.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle
const char **	restrictions	Input	This works as a restriction, and denotes a string containing 3 array elements.

3.25.3 Return Values

altibase_list_fields() returns result set for success, or null if an error occurred.

3.25.4 Description

altibase_list_fields() returns result set consisting of field names applied to requests that meet the coditions. A string should contain 3 array elements. If there are more than 3 array elements, the rest are ignored execept 3 array elements from the first. The following array elements work as a restriction.

Index	Condition	Meaning
0	User name	This is the LIKE condition which allows you to retrieve information of user name. If a user name argument is set to null or ALTIBASE_ALL_USERS, all privileges are granted on a user name argument.
1	Table name	This is the LIKE condition which allows you to retrieve information of table name. If a table name argument is set to null or ALTIBASE_ALL_TABLES, all privileges are granted on a table name argument.

Index	Condition	Meaning
2	Column name	This is the LIKE condition which allows you to retrieve information of column name. If a column name argument is set to null or ALTIBASE_ALL_COLUMNS, all privileges are granted on a column name argument.

The values of arguments are same as those used in LIKE condition. See *SQL Reference* to know how to perform patttern matching including '%' or '_'.

If you do not want to place the restriction, you can specify an argument as null, ALTIBASE_ALL_USERS, ALTIBASE_ALL_TABLES, or ALTIBASE_ALL_COLUMNS. Restriction argument must not contain null. One of restriction arguments should be valid at least.

You cannot call altibase_list_fields() while executiong other queries, or you cannot run other queries while using result set by calling this function. Result set contains columns as follows.

Column Name	Column Number	Data Type	Description
TABLE_CAT	1	VARCHAR	This field contains the catalog name of the table, and always returns null.
TABLE_SCHEM	2	VARCHAR	This field contains the schema name of the table. If this is not appropriate for database, this returns null.
TABLE_NAME	3	VARCHAR (NOT NULL)	This field contains the name of the table.
COLUMN_NAME	4	VARCHAR (NOT NULL)	This field contains the name of the column. If column is not defined, it returns an empty string,
DATA_TYPE	5	SMALLINT (NOT NULL)	This field contains SQL data type.
TYPE_NAME	6	VARCHAR (NOT NULL)	This field contains a character string which represents the name of the data type corresponding to DATA_TYPE.
COLUMN_SIZE	7	INTEGER	This field contains the size of the column. If this is not appropriate for database, this returns null.
BUFFER_LENGTH	8	INTEGER	This field denotes maximum buffer length to store the data.
DECIMAL_DIGITS	9	SMALLINT	This field denotes number of decimal digits stored in the column. If this cannot be applied to data type, this returns null.

Column Name	Column Number	Data Type	Description
NUM_PREC_RADIX	10	SMALLINT	If the column has a decimal numeric type and NUM_PREC_RADIX has the value 10, COLUMN_SIZE and DECIMAL_DIGITS have values which are decimal numbers allowed in the columnn. For example, a DECIMAL value is defined as DECIMAL(12, 5), this indicates that NUM_PREC_RADIX has the value 10, COLUMN_SIZE has the value 12, and DECIMAL_DIGITS has the value 5.
NULLABLE	11	SMALLINT (NOT NULL)	This field indicates if null values can be ever supported. If they can, this returns 1. Otherwise, this returns 0.
REMARKS	12	VARCHAR	This field contains a description of the column in the table.
COLUMN_DEF	13	VARCHAR	This field indicates default value of the column, and can be used to initialize the table.
SQL_DATA_TYPE	14	SMALLINT (NOT NULL)	This field contains SQL data type.
SQL_DATETIME_SUB	15	SMALLINT	This field returns an integer value representing a datetime subtype code, or null for SQL data types to which this does not apply.
CHAR_OCTET_LENGTH	16	INTEGER	This field returns maximum number of digits for character or binary string, or null for other data types.
ORIGINAL_POSITION	17	INTEGER (NOT NULL)	This fiedl indicates column order of the table. The column number starts at offset 1.
IS_NULLABLE	18	VARCHAR	NO : nulls are not included in the column. YES : nulls are included in the column.
STORE_TYPE	19	CHAR(1)	This field determines the type of column to store. V: A column is stored in variable length format. F: A column is stored in fixed length format. L: A column is stored in LOB format.

The results are aligned by using TABLE_CAT, TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION. altibase_list_fields() cannot be used with the functions such as altibase_use_result() and altibase_list_tables() which return result set. You must free current result set handle with altibase_free_result() to obtain other one.

3.26 altibase_list_tables()

altibase_list_tables() returns a result set consisting of table names in the current database.

3.26.1 Syntax

```
ALTIBASE_RES altibase_list_tables
(
    ALTIBASE    altibase,
    const char *restrictions[]
)
```

3.26.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle
const char **	restrictions	Input	This works as a restriction, and denotes a string containing 3 array elements.

3.26.3 Return Value

altibase_list_tables() returns result set for success, or null if an error occurred.

3.26.4 Description

altibase_list_tables() returns a result set consisting of table names applied to requests that meet the conditions. A string should contain 3 array elements. If there are more than 3 array elements, the rest are ignored except 3 array elements from the first. The following array elements work as a restriction.

Index	Condition	Meaning
0	User name	This is the LIKE condition which allows you to retrieve information of user name. If a user name argument is set to null or ALTIBASE_ALL_USERS, all privileges are granted on a user name argument.
1	Table name	This is the LIKE condition which allows you to retrieve information of table name. If a table name argument is set to null or ALTIBASE_ALL_TABLES, all privileges are granted on a table name argument.

Index	Condition	Meaning
2	Table type	This is the LIKE condition which allows you to retrieve information of column name. If a column name argument is set to null or ALTIBASE_ALL_TABLE_TYPES, all privileges are granted on a column name argument.

The values of arguments are same as those used in LIKE condition. See *SQL Reference* to know how to perform patttern matching including '%' or '_'. If you do not want to place the restriction, you can specify an argument as null, ALTIBASE_ALL_USERS, ALTIBASE_ALL_TABLES, or ALTIBASE_ALL_TABLE_TYPES. Restriction argument must not contain null. One of restriction arguments should be valid at least.

You cannot call altibase_list_tables() while executiong other queries, or you cannot run other queries while using result set by calling this function. Result set contains columns as follows.

Column Name	Column Number	Data Type	Description
TABLE_CAT	1	VARCHAR	This field contains the catalog name of the table, and always returns null.
TABLE_SCHEM	2	VARCHAR	This field contains the schema name of the table. If this is not appropriate for database, this returns null.
TABLE_NAME	3	VARCHAR (NOT NULL)	This field contains the name of the table.
TABLE_TYPE	4	VARCHAR	This field denotes the table type. (Only TABLE exists in Altibase.)
REMARK	5	VARCHAR	This field is not enabled.
MAXROW	6	BIGINT	This field represents the maximum number of rows a result set can contain. If this is set to 0, the number of rows is not limited.
TABLESPACE_NAME	7	VARCHAR(512)	This field represents the name of the tablespace.
TABLESPACE_TYPE	8	INTEGER	This field represents the type of the tablespace.
PCTFREE	9	INTEGER	This field is used to sepcify how much space should be left in the page for updates.
PCTUSED	10	INTEGER	This field represents the percent of used space that Altibase maintains for each data page.

The results are aligned by using TABLE_TYPE, TABLE_CAT, TABLE_SCHEM and TABLE_NAME. altibase_list_tables() cannot be used with the functions such as altibase_use_result() and

3.26 altibase_list_tables()

altibase_list_tables() which return result set. You must free current result set handle with altibase_free_result() to obtain other one.

3.27 altibase_next_result()

altibase_next_result() moves the cursor position on the next statement result set to read.

3.27.1 Syntax

```
int altibase_next_result
(
    ALTIBASE altibase
)
```

3.27.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.27.3 Return Value

Return Value	Description
ALTIBASE_SUCCESS	Successful and there are more result sets
ALTIBASE_NO_DATA	Successful and there are no more result sets
ALTIBASE_ERROR	An error occurred.

3.27.4 Description

This function is used when you execute multiple statements and then read the next statement result set. Before each call to altibase_next_result(), you must call altibase_free_result() for the current statement if it is a statement that returned a result set.

After calling altibase_next_result(), the state of the connection is as if you had called altibase_query() for the next statement. This means that you can call altibase_store_result() and altibase_affected_rows().

3.28 altibase_num_fields()

altibase_num_fields() returns the number of columns in a result set.

3.28.1 Syntax

```
int altibase_num_fields
(
    ALTIBASE_RES result
)
```

3.28.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_RES	result	Input	Resukt handle

3.28.3 Return Value

altibase_num_fields() returns the number of columns in a result set for success, or ALTIBASE_INVALID_FIELDCOUNT if an error occurred.

3.28.4 Description

altibase_num_fields() retrieves the number of columns from a result set. You can get the number of columns either from a pointer to a result set or to a connection handle. You would use the connection handle if altibase_store_result() or altibase_use_result() returned null. You can call altibase_field_count() when using connection handle to get the number of columns.

3.29 altibase_num_rows()

altibase_num_rows() returns the number of rows in the result set.

3.29.1 Syntax

```
ALTIBASE_LONG altibase_num_rows
(
    ALTIBASE_RES result
)
```

3.29.2 Argument

Data Type	Argument	In/Out	Description
ALTIBASE_RES	result	Input	Result handle

3.29.3 Return Value

altibase_num_rows() returns the number of rows in the result set.

3.29.4 Description

altibase_num_rows() retrives the number of rows from a result set. The use of altibase_num_rows() depends on whether you use altibase_store_result() or altibase_use_result() to return the result set.

If you use altibase_store_result(), altibase_num_rows() returns the correct value. However, if you use altibase_num_rows(), altibase_num_rows() does not return the correct value until all the rows in the result set have been retrieved.

altibase_num_rows() is intended fro use with statements that return a result set such as SELECT. For statements such as INSERT, UPDATE and DELETE, the number of affected rows can be obtained with altibase_affected_rows().

3.30 altibase_proto_version()

altibase_proto_version() returns a constant representing the protocol version used by the current connection.

3.30.1 Syntax

```
int altibase_proto_version
(
    ALTIBASE altibase
)
```

3.30.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.30.3 Return Values

altibase_proto_version() returns the protocol version used by the current connection as a constant on success, or ALTIBASE_INVALID_VERSION if connection handle is not valid or connection is closed.

3.30.4 Description

altibase_proto_version() returns a constant representing the protocol version used by the current connection. The value has the format `Mmmddpppp` whose specific meaning is as follows.

Format	Meaning	Note
M	Where `M` is the major version.	
mm	Where `mm` is the minor version.	When you assign a value to `mm`, if the value is shorter than the declared length of this, Altibase pads 0 to the rest space.
dd	Where `dd` is the development version.	This is always set to 0.
pppp	Where `pppp` is the patch level.	When you assign a value to `pppp`, if the value is shorter than the declared length of this, Altibase pads 0 to the rest space.

For example, a value of 506000001 represents the protocol verison used by the current connection of 5.6.0.1.

3.31 altibase_proto_verstr()

altibase_proto_verstr() returns a string representing the protocol version used by the current connection.

3.31.1 Syntax

```
const char * altibase_proto_verstr
(
    ALTIBASE altibase
)
```

3.31.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.31.3 Return Values

altibase_proto_verstr() returns the protocol version used by the current connection as a string on success, or null if connection handle for a string is not valid or connection is closed.

3.31.4 Description

altibase_proto_verstr() returns a string representing the protocol version used by the current connection. The value has the format x.x.0.x. You must not change or release it as you please because it is manged within procedure.

3.32 altibase_server_version()

altibase_server_version() returns the version number of the server.

3.32.1 Syntax

```
int altibase_server_version
(
    ALTIBASE altibase
)
```

3.32.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.32.3 Return Values

altibase_server_version() returns the number that represents the Altibase server version on success, or ALTIBASE_INVALID_VERSION if connection handle is not valid, connection is closed or the function fails to return the value.

3.32.4 Description

altibase_server_version() returns the version number of the server as an integer. The value has the format `Mmmddpppp` whose meaning is as follows.

Format	Meaning	Note
M	Where `M` is the major version.	
mm	Where `mm` is the minor version.	When you assign a value to `mm`, if the value is shorter than the declared length of this, Altibase pads 0 to the rest space.
dd	Where `dd` is the development version.	When you assign a value to `dd`, if the value is shorter than the declared length of this, Altibase pads 0 to the rest space.
pppp	Where `pppp` is the patch level.	When you assign a value to `pppp`, if the value is shorter than the declared length of this, Altibase pads 0 to the rest space.

For example, a value of 503050001 represents the version number of the server of 5.3.5.1.

3.33 altibase_server_verstr()

altibase_server_verstr() returns a string that represents the server version number.

3.33.1 Syntax

```
const char * altibase_server_verstr
(
    ALTIBASE altibase
)
```

3.33.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

3.33.3 Return Values

altibase_server_verstr() returns a character string that represents the server version number on success, or null if connection handle is not valid, connection is closed, or the function fails to return the value.

3.33.4 Description

altibase_server_verstr() returns the server version number as s character string. The value has the format x.x.x.x. You must not change or release it as you please because it is manged within procedure.

3.34 altibase_set_charset()

altibase_set_charset() is used to set the character set for the current connection.

3.34.1 Syntax

```
int altibase_set_charset
(
    ALTIBASE    altibase,
    const char *csname
)
```

3.34.2 Arguments

Data Type	Arguments	In/Out	Description
ALTIBASE	altibase	Input	Connection handle
const char *	csname	Input	Character set name

3.34.3 Return Values

altibase_set_charset() returns ALTIBASE_SUCCESS for success, or ALTIBASE_ERROR if an error occurred.

3.34.4 Description

altibase_set_charset() is used to set the character set for the current connection. You must call this function to specify the character set before connection. Additionally the character set can be derived from NLS_USE environment variable or connection string. Its default is derived from environment variable.

3.34.5 Example

```
ALTIBASE altibase;

altibase = altibase_init();
/* ... check return value ... */

rc = altibase_set_charset(altibase, "KO16KSC5601"));
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    /* ... error handling ... */
}

rc = altibase_connect(altibase, CONNSTR);
/* ... check return value ... */
```

3.35 altibase_set_failover_callback()

altibase_set_failover_callback() registers failover callbcaks for the failover to happen in Altibase.

3.35.1 Syntax

```
int altibase_set_failover_callback
(
    ALTIBASE                     altibase,
    ALTIBASE_FAILOVER_CALLBACK   callback,
    void                        *app_context
)
```

3.35.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle
ALTIBASE_FAILOVER_CALLBACK	callback	Input	This denotes failover callback for registeration. If this is set to null, you can cancel the registeration.
void *	app_context	Input	This denotes user context. This is also function pointer used by callback to store an address of a function.

3.35.3 Returned Values

altibase_set_failover_callback() returns ALTIBASE_SUCCESS for success, or ALTIBASE_ERROR if an error occurred.

3.35.4 Description

altibase_set_failover_callback() need to be called to register failover callbcaks for communication with user application only at the time of STF(Service Time Failover). If you want to cancel the registeration, the callback argument should be set to null. You must register failover callbacks after calling altibase_connect() successfully.

3.35.5 Example

See the example in *chapter 4. Fail-Over* of *Replication Manual*.

3.36 altibase_set_option()

altibase_set_option() enables or disables an option for the connection.

3.36.1 Syntax

```
int altibase_set_option
(
    ALTIBASE         altibase,
    ALTIBASE_OPTION  option,
    const void      *arg
)
```

3.36.2 Argument

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle
ALTIBASE_OPTION	option	Input	Option type
const void *	arg	Input	Option value

3.36.3 Return Values

altibase_set_option() returns ALTIBASE_SUCCESS for success, or ALTIBASE_ERROR if an error occurred.

3.36.4 Description

altibase_set_option() enables or disables an option for the connection. You can call this function many times when enabling several options. altibase_set_option() can be used after calling altibase_init() and before calling altibase_connect(). For details about an option for connection, see enum ALTIBASE_OPTION.

3.36.5 Example

```
ALTIBASE altibase;

altibase = altibase_init();
/* ... check return value ... */

rc = altibase_set_option(altibase, ALTIBASE_APP_INFO, "myapp");
/* ... check return value ... */
rc = altibase_set_option(altibase, ALTIBASE_NLS_USE, "KO16KSC5601");
/* ... check return value ... */
```

```
rc = altibase_connect(altibase, CONNSTR);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    fprintf(stderr, "Failed to connect: %s\n", altibase_error(altibase));
}
```

3.36 altibase_set_option()

4 Prepared Statement Function Descriptions

This chapter describes the functions available for prepared statement processing by using state-ment handle in great detail.

See *Error Message Reference* for error messages related to using these functions.

4.1 altibase_stmt_affected_rows()

altibase_stmt_affected_rows() may be called immediately after executing a statement. It returns the number of rows changed, deleted, or inserted by the last statement if it was an UPDATE, DELETE, or INSERT. It is like altibase_affected_rows() but for prepared statements.

4.1.1 Syntax

```
ALTIBASE_LONG altibase_stmt_affected_rows
(
    ALTIBASE_STMT stmt
)
```

4.1.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	altibase	Input	Statment Handle

4.1.3 Return Values

Return Value	Description
Greater than 0	An integer indicates the number of rows affected or retrieved.
0	Zero indicates that no rows were updated or that no query has yet been executed.
ALTIBASE_INVALID_AFFECTEDROWS	This indicates that the query returned an error.

4.1.4 Description

altibase_stmt_affected_rows() returns the value that it would return for the last statement executed within the procedure. For UPDATE statements, the affected-rows value by default is the number of rows actually changed. For DELETE statements, the affected-rows value is the number of deleted rows. For INSERT statements, the affected-rows value is the number of existing rows which are updated. For SELECT statements, the affected-rows value is 0, and altibase_stmt_affected_rows() works like altibase_num_rows() which returns the number of rows selected by a SELECT statement.

4.1.5 Example

```
char *qstr = "UPDATE t1 SET val = val * 1.1 WHERE type = 1";

rc = altibase_stmt_prepare(stmt, qstr);
/* ... check return value ... */

rc = altibase_stmt_execute(stmt);
/* ... check return value ... */

printf("%ld updated\n", altibase_stmt_affected_rows(stmt));
```

4.2 altibase_stmt_bind_param()

altibase_stmt_bind_param() is used to bind input data for the parameter markers in the SQL statement.

4.2.1 Syntax

```
int altibase_stmt_bind_param
(
    ALTIBASE_STMT  stmt,
    ALTIBASE_BIND *bind
)
```

4.2.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle
ALTIBASE_BIND *	bind	Input	altibase_stmt_bind_param() uses ALTIBASE_BIND structure to supply the data. The bind argument is the address of an array of ALTIBASE_BIND structures.

4.2.3 Return Values

altibase_stmt_bind_param() returns ALTIBASE_SUCCESS if the bind operation was successful, or ALTIBASE_ERROR if an error occurred.

4.2.4 Description

altibase_stmt_bind_param() binds variables to a prepared statement as the parameter marker in the SQL statement. The values of parameter markers are substitued for the question marks in SQL statement.

The client library expects the array to contain one element for each "?"parameter marker that is present in the query. If three parameter markers are declared, the array of ALTIBASE_BIND structures must contain three elements.

The bind argument is available before you call altibase_stmt_reset(), altibase_stmt_close() or altibase_close(). Therefore, if inputting several data in the SQL statement, you can substitue the bound data for the values and then call altibase_stmt_execute() multiple times after passing the SQL statement to altibase_stmt_prepare() and altibase_stmt_bind_param().
altibase_stmt_bind_param() must be called after calling altibase_stmt_prepare() and altibase_stmt_set_array_bind(), and before calling altibase_stmt_execute().

4.2.5 Examples

```
#define PARAM_COUNT 2
#define STR_SIZE    50
#define QSTR        "INSERT INTO t1 VALUES (?, ?)"

int           int_dat;
char          str_dat[STR_SIZE];
ALTIBASE_LONG length[PARAM_COUNT];

ALTIBASE      altibase;
ALTIBASE_STMT stmt;
ALTIBASE_BIND bind[PARAM_COUNT];
int           rc;
int           i;

/* ... omit ... */

int_dat = 1;
strcpy(str_dat, "test1");

length[0] = sizeof(int);
length[1] = ALTIBASE_NTS;

memset(bind, 0, sizeof(bind));

bind[0].buffer_type   = ALTIBASE_BIND_INTEGER;
bind[0].buffer        = &int_dat;
bind[0].length        = &length[0];

bind[1].buffer_type   = ALTIBASE_BIND_STRING;
bind[1].buffer        = str_dat;
bind[1].buffer_length = STR_SIZE;
bind[1].length        = &length[1];

stmt = altibase_stmt_init(altibase);
/* ... check return value ... */

rc = altibase_stmt_prepare(stmt, QSTR);
/* ... check return value ... */

rc = altibase_stmt_bind_param(stmt, bind);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    for (i = 0; i < PARAM_COUNT; i++)
    {
        printf("bind %d : %d\n", i, bind[i].error);
    }
    /* ... error handling ... */
}

rc = altibase_stmt_execute(stmt);
/* ... check return value ... */
nt atomic_array = ALTIBASE_ATOMIC_ARRAY_ON;

rc = altibase_stmt_set_attr(stmt, ALTIBASE_STMT_ATTR_ATOMIC_ARRAY,
                            &atomic_array);
/* ... check return value ... */

rc = altibase_stmt_get_attr(stmt, ALTIBASE_STMT_ATTR_ATOMIC_ARRAY,
                            &atomic_array);
/* ... check return value ... */

printf("Atomic array : %d\n", atomic_array);
```

4.3 altibase_stmt_execute()

altibase_stmt_execute() executes the prepared query associated with the statement handle.

4.3.1 Syntax

```
int altibase_stmt_execute
(
    ALTIBASE_STMT stmt
)
```

4.3.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.3.3 Return Value

Return Value	Description
ALTIBASE_SUCCESS	Execution was successful.
ALTIBASE_NEED_DATA	There are data which you want to send by calling altibase_stmt_send_long_data().
ALTIBASE_ERROR	An error occurred.

4.3.4 Description

altibase_stmt_execute() executes the prepared query associated with the statement handle. For an UPDATE, DELETE or INSERT, the number of changed, deleted, or inserted rows can be found by calling altibase_stmt_affected_rows(). For a statement such as SELECT that generates a result set, you must call altibase_stmt_fetch() to fetch the data. You must call altibase_stmt_free_result() to free a result set after using it.

4.3.5 Example

See the examples in altibase_stmt_bind_param() and altibase_stmt_bind_result().

4.4 altibase_stmt_bind_result()

altibase_stmt_bind_result() is used to bind output columns in the result set.

4.4.1 Syntax

```
int altibase_stmt_bind_result
(
    ALTIBASE_STMT  stmt,
    ALTIBASE_BIND *bind
)
```

4.4.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	Stmt	Input	Statement handle
ALTIBASE_BIND *	bind	Input	The bind argument should be set to receive output values.

4.4.3 Return Values

altibase_stmt_bind_result() returns ALTIBASE_SUCCESS if the bind operation was successful, or ALTIBASE_ERROR if an error occurred.

4.4.4 Description

altibase_stmt_bind_result() binds variables to a prepared statement for result storage. The client library expects the array to contain one element for each column of the result set. If three parameter markers are declared, the array of ALTIBASE_BIND structures must contain three elements.

The bind argument is available before you call altibase_stmt_reset(), altibase_stmt_close() or altibase_close(). When altibase_stmt_fetch() is called to fetch data, the Altibase protocol places the data for the bound columns into the specified buffers. Therefore, you can know the values returned by altibase_stmt_bind_result() in this way. altibase_stmt_bind_result() must be called after calling altibase_stmt_prepare() and altibase_stmt_set_array_fetch(), and before calling altibase_stmt_store_result() or altibase_stmt_fetch().

4.4.5 Example

```
#define FIELD_COUNT 2
#define STR_SIZE    50
#define QSTR        "SELECT * FROM t1"

ALTIBASE       altibase;
```

```
ALTIBASE_STMT stmt;
ALTIBASE_BIND bind[FIELD_COUNT];
int           int_dat;
char          str_dat[STR_SIZE];
ALTIBASE_LONG length[FIELD_COUNT];
ALTIBASE_BOOL is_null[FIELD_COUNT];
int           rc;
int           row;

/* ... omit ... */

stmt = altibase_stmt_init(altibase);
/* ... check return value ... */

rc = altibase_stmt_prepare(stmt, QSTR);
/* ... check return value ... */

rc = altibase_stmt_execute(stmt);
/* ... check return value ... */

memset(bind, 0, sizeof(bind));

bind[0].buffer_type   = ALTIBASE_BIND_INTEGER;
bind[0].buffer        = &int_dat;
bind[0].length        = &length[0];
bind[0].is_null       = &is_null[0];

bind[1].buffer_type   = ALTIBASE_BIND_STRING;
bind[1].buffer        = str_dat;
bind[1].buffer_length = STR_SIZE;
bind[1].length        = &length[1];
bind[1].is_null       = &is_null[1];

rc = altibase_stmt_bind_result(stmt, bind);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    for (i = 0; i < FIELD_COUNT; i++)
    {
        printf("bind %d : %d\n", i, bind[i].error);
    }
    /* ... error handling ... */
}

/* altibase_stmt_store_result() is optional */
rc = altibase_stmt_store_result(stmt);
/* ... check return value ... */

for (row = 0; (rc = altibase_stmt_fetch(stmt)) != ALTIBASE_NO_DATA; row++)
{
    if (ALTIBASE_NOT_SUCCEEDED(rc))
    {
        /* ... error handling ... */
        break;
    }

    printf("row %d : ", row);
    if (is_null[0] == ALTIBASE_TRUE)
    {
        printf("{null}");
    }
    else
    {
        printf("%d", int_dat);
    }
    printf(", ");
    if (is_null[1] == ALTIBASE_TRUE)
    {
        printf("{null}");
    }
    else
    {
        printf("(%d) %s", length[1], str_dat);
    }
    printf("\n");
}

rc = altibase_stmt_free_result(stmt);
/* ... check return value ... */
```

4.5 altibase_stmt_data_seek()

altibase_stmt_data_seek() seeks to an arbitrary row in a statememt result set and moves its position.

4.5.1 Syntax

```
int altibase_stmt_data_seek
(
    ALTIBASE_STMT stmt,
    ALTIBASE_LONG offset
)
```

4.5.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle
ALTIBASE_LONG	offset	Input	This is a row number which starts at 0.

4.5.3 Return Values

altibase_stmt_data_seek() returns ALTIBASE_SUCCESS if successful, or ALTIBASE_ERROR if an error occurred.

4.5.4 Description

altibase_stmt_data_seek() moves the row position in a stateiment result set to the specified place, The offset value is a row number and should be in the range from 0 to altibase_stmt_num_rows(stmt) - 1. altibase_stmt_data_seek() may be used only in conjunction with altibase_stimt_store_result().

4.5.5 Examples

```
#define QSTR "SELECT last_name, first_name FROM friends"

/* ... omit ... */

rc = altibase_stmt_store_result(stmt);
/* ... check return value ... */

row_count = altibase_stmt_num_rows(stmt);
for (i = 0; i < row_count; i++)
{
    rc = altibase_stmt_data_seek(stmt, i);
    if (ALTIBASE_NOT_SUCCEEDED(rc))
    {
        printf("ERR : %d : ", i, altibase_error());
        continue;
    }

    rc = altibase_stmt_fetch(stmt);
    /* ... check return value ... */

    /* ... omit ... */
}

rc = altibase_stmt_free_result(stmt);
/* ... check return value ... */
```

4.6 altibase_stmt_errno()

altibase_stmt_errno() returns the error code for the most recently invoked statement.

4.6.1 Syntax

```
unsigned int altibase_stmt_errno
(
    ALTIBASE_STMT stmt
)
```

4.6.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.6.3 Result Values

altibase_stmt_errno() retunrs 0 if the most recently invoked statement was successful and no error occurred. The function returns an error code value if an error occurred.

4.6.4 Description

altibase_stmt_errno() returns the error code for the most recently invoked statemenf function that can fail. All functuins do not return error codes. Error codes are returned by queries for their operation. Errors are listed at Error Message Refrence in detail. Make sure you check the value before calling another function because it is initialized or new one is created instead if you call another function. The value returned by altibase_stmt_errno() is different from that of SQLSTATE. You should use altibase_sqlstate() to find a specific SQLSTATE when handling errors. It is recommanded not to check the values returned by altibase_errno() but those of SQLSTATE if you need error code.

4.6.5 Example

```
rc = altibase_stmt_execute(stmt);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    printf("error no  : %05X\n", altibase_stmt_errno(stmt));
    printf("error msg : %s\n", altibase_stmt_error(stmt));
    printf("sqlstate  : %s\n", altibase_stmt_sqlstate(stmt));
    return 1;
}

/* ... omit ... */
```

4.7 altibase_stmt_error()

altibase_stmt_error() returns error message for the most recently invoked statement.

4.7.1 Syntax

```
const char * altibase_stmt_error
(
    ALTIBASE_STMT stmt
)
```

4.7.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.7.3 Result Values

altibase_error() returns the error text from the last function, or an empty string if no error occurred.

4.7.4 Description

If the ost recently invoked statement fails, altibase_stmt_error() returns the error message. Otherwise, the function returns an empty string.

Make sure you check the value before calling another function because it is initialized or new one is created instead if you call another function. You must not change or release it as you please because it is manged within procedure.

4.7.5 Example

See the example in altibase_stmt_errno().

4.8 altibase_stmt_fetch()

altibase_stmt_fetch() fetches a row from the result set in a prepared statement.

4.8.1 Syntax

```
int altibase_stmt_fetch
(
    ALTIBASE_STMT stmt
)
```

4.8.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.8.3 Return Value

Return Value	Description
ALTIBASE_SUCCESS	Successful, the data has been fetched.
ALTIBASE_SUCCESS_WITH_INFO	The data has been fetched. However, error also occurred.
ALTIBASE_NO_DATA	No more data exists.
ALTIBASE_ERROR	An error occurred.

4.8.4 Description

altibase_stmt_fetch() returns a row data from the result set in a prepared statement using the buffers bound.

4.8.5 Example

See the example in altibase_stmt_bind_result().

4.9 altibase_stmt_fetch_column()

altibase_stmt_fetch_column() fetches one column from the current result set row.

4.9.1 Syntax

```
int altibase_stmt_fetch_column
(
    ALTIBASE_STMT  stmt,
    ALTIBASE_BIND *bind,
    int            column,
    ALTIBASE_LONG  offset
)
```

4.9.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle
ALTIBASE_BIND *	bind	Input/Output	This denotes a buffer storing data.
int	column	Input	This is the number of a returned column. Its value starts at 0.
ALTIBASE_LONG	offset	Input	This is the offset value which starts at 0.

4.9.3 Return Value

altibase_stmt_fetch_columns() returns ALTIBASE_SUCCESS in case of success, or ALTIBASE_ERROR if an error occurred.

4.9.4 Description

altibase_stmt_fetch_column() returns one column from the current result set row to the bind argument. The offset argument is the offset within the data value at which to begin retrieving data. This can be used for fetching the data value in pieces. The beginning of the value is offset 0.

If the offset argument is set to ALTIBASE_FETCH_CONT, altibase_stmt_fetch_column() returns columns which are placed after the returned one previously. If altibase_stmt_fetch_column() has not returned one column before, the function returns it at the starting position. You can use altibase_stmt_fetch_column() differently depending on calling altibase_stmt_store_result().

altibase_stmt_store_result()	The bind argument	The offset argument
The function is called.	The value of its buffer_type should be same as that returned by altibase_stmt_bind_result().	The offset argument can have a random value.
The function is not called.	Its buffer_type can have a random value.	The offset argument should be set to ALTIBSE_FETCH_CONT for the value returned in order.

An error occurs if the bind and offset arguments do not meet requirements of their restrictions.

4.9.5 Example

```
#define STR_SIZE 50

char           str_dat[STR_SIZE];
ALTIBASE_LONG  length;
ALTIBASE_BOOL  is_null;
ALTIBASE_BIND  bind;
int            rc;
int            i;

/* ... omit ... */

rc = altibase_stmt_execute(stmt);
/* ... check return value ... */

memset(bind, 0, sizeof(bind));

bind.buffer_type   = ALTIBASE_BIND_STRING;
bind.buffer        = str_dat;
bind.buffer_length = STR_SIZE;
bind.length        = &length;
bind.is_null       = &is_null;

while (1)
{
    rc = altibase_stmt_fetch(stmt);
    if (rc == ALTIBASE_NO_DATA)
    {
        break;
    }
    if (ALTIBASE_NOT_SUCCEEDED(rc))
    {
        /* ... error handling ... */
    }

    for (i = 0; ; i++)
    {
        rc = altibase_stmt_fetch_column(stmt, &bind, 0, ALTIBASE_FETCH_CONT);
        if (ALTIBASE_NOT_SUCCEEDED(rc))
        {
            /* ... error handling ... */
        }
        printf("%d : (%d) %s\n", i, length, str_dat);
    }
}
```

4.10 altibase_stmt_fetched()

altibase_stmt_fetched() returns the number of the exisitng rows fetched previously after fetching new result as an array.

4.10.1 Syntax

```
ALTIBASE_LONG altibase_stmt_fetched
(
    ALTIBASE_STMT stmt
)
```

4.10.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.10.3 Return Value

altibase_stmt_fetched() returns the number of the exisitng rows fetched previously after fetching new result as an array in case of success, or ALTIBASE_INVALID_FETCHED if an error occurred.

4.10.4 Description

altibase_stmt_fetched() returns the number of the exisitng rows fetched previously only after fetching new result as an array.

4.10.5 Example

See 5.3 Array Fetching in Chapter5.Using Array Binding and Array Fetching.

4.11 altibase_stmt_num_rows()

altibase_stmt_num_rows() returns the number of rows in the result set.

4.11.1 Syntax

```
ALTIBASE_LONG altibase_stmt_num_rows
(
    ALTIBASE_STMT stmt
)
```

4.11.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.11.3 Return Value

altibase_stmt_num_rows() returns the number of rows in the result set.

4.11.4 Description

altibase_stmt_num_rows() returns the number of rows in the result set. The use of altibase_stmt_num_rows() depends on whether you used altibase_stmt_store_result() to buffer the entire result set. If you use altibase_stmt_store_result(), altibase_stmt_num_rosw() may be called immediately. Otherwise, the row count is unavailable unless you count the rows as you fetch them.

altibase_stmt_num_rows() is intended for use with statements that return a result set, such as SELECT. For statements such as INSERT, UPDATE, or DELETE, the number of affected rows can be obtained with altibase_stmt_affected_rows().

4.11.5 Example

See the example in altibase_stmt_data_seek().

4.12 altibase_stmt_sqlstate()

altibase_stmt_sqlstate() returns a null-terminating string containing the SQLSTATE error code for the most recently invoked prepared statement function that can succeed or fail.

4.12.1 Syntax

```
const char * altibase_stmt_sqlstate
(
    ALTIBASE_STMT stmt
)
```

4.12.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.12.3 Return Value

altibase_stmt_sqlstate() returns a null-terminating character string containing the SQLSTATE error code in case of success.

4.12.4 Description

altibase_stmt_sqlstate() returns a null-terminating string containing the SQLSTATE error code for the most recently invoked prepared statement function that can succeed of fail. The error code consists of five characters. “00000” means “no error”. For a list of possible values, see *Altibase Error Message Reference*.

Make sure you check the value before calling another function becuase it is initialized or new one is created instead if you call another function. The value returned by altobase_stmt_errno() is different from that of SQLSTATE. You should use altobase_sqlstate() to find a specific SQLSTATE when handling errors. It is recommended not to check the values returned by altibase_stmt_errno() but those of SQLSTATE if you need error code.

Not all Altibase error number returned by altibase_stmt_errno() are mapped to SQLSTATE error codes. Therefore, you cannot know the values of SQLSTATE by checking those returned by altibase_stmt_errno(), or you cannot know the values returned by altibase_stmt_errno() by checking those of SQLSTATE exactly. You must not change or cancel it as you please because it is managed within procedure.

4.12.5 Example

See the example in altibase_stmt_errno().

4.13 altibase_stmt_store_result()

altibase_stmt_store_result() is called to buffer the complete result set.

4.13.1 Syntax

```
int altibase_stmt_store_result
(
    ALTIBASE_STMT stmt
)
```

4.13.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.13.3 Return Value

altibase_stmt_store_result() returns ALTIBASE_SUCCESS if the results are buffered, or ALTIBASE_ERROR if an error occurred.

4.13.4 Description

altibase_stmt_store_result() transfers the complete result set from a prepared statement. The complete result set is buffered on the client from the server by calling the function.

Even though you call altibase_stmt_fetch(), there is no communication with the server because the complete result is already buffered. Great attention should be paid to call altibase_stmt_store_result() because there can be insufficient memory if result set contains large amounts of data such as LOB or geometry.

altbase_stmt_store_result() can be used after calling altibase_stmt_bind_result() and before calling altibase_stmt_fetch(). If calling altibase_stmt_store_result(), you can use the followings additionally.

- altibase_stmt_num_rows()
- altibase_stmt_data_seek()

You must call altibase_stmt_free() to free current result set after you are done with the result set.

4.13.5 Example

See the examples in altibase_stmt_bind_result() and altibase_stmt_data_seek().

4.14 altibase_stmt_close()

altibase_stmt_close() closes the prepared statement.

4.14.1 Syntax

```
int altibase_stmt_close
(
    ALTIBASE_STMT stmt
)
```

4.14.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	Stmt	Input	Statement handle

4.14.3 Return Values

altibase_stmt_close() returns ALTIBASE_SUCCESS if the statement was freed successfully, or ALTIBASE_ERROR if an error occurred.

4.14.4 Description

altibase_stmt_close() closes the prepared statement and also deallocates the statement handle. The function removes entire resources allocated to connection handle.

4.14.5 Example

See the example in altibase_stmt_init().

4.15 altibase_stmt_field_count()

altibase_stmt_field_count() returns the number of columns for the most recent statement for the statement handler.

4.15.1 Syntax

```
int altibase_stmt_field_count
(
    ALTIBASE_STMT stmt
)
```

4.15.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.15.3 Return Value

Return Value	Description
Greater than 0	An integer indicates the number of columns for the most recent statement for the statement handler.
0	Zero indicates the statement which do not return a result set.
ALTIBASE_INVALID_FIELDCOUNT	This indicates that an error occurred.

4.15.4 Description

altibase_stmt_field_count() returns the number of columns for the most recent statement for the statement handler. The function returns 0 for statements such as INSERT, DELETE and UPDATE which do not return a result set. altibase_stmt_field_count() can be called after you have prepared a statement by invoking ().

4.15.5 Example

See the example in altibase_stmt_prepare().

4.16 altibase_stmt_free_result()

altibase_stmt_field_count() returns the number of columns for the most recent statement for the statement handler.

4.16.1 Syntax

```
int altibase_stmt_field_count
(
    ALTIBASE_STMT stmt
)
```

4.16.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.16.3 Return Value

Return Value	Description
Greater than 0	An integer indicates the number of columns for the most recent statement for the statement handler.
0	Zero indicates the statement which do not return a result set.
ALTIBASE_INVALID_FIELDCOUNT	This indicates that an error occurred.

4.16.4 Description

altibase_stmt_field_count() returns the number of columns for the most recent statement for the statement handler. The function returns 0 for statements such as INSERT, DELETE and UPDATE which do not return a result set. altibase_stmt_field_count() can be called after you have prepared a statement by invoking altibase_stmt_prepare().

4.16.5 Example

See the example in altibase_stmt_prepare().

4.17 altibase_stmt_param_count()

altibase_stmt_param_count() returns the number of parameter markers present in the prepared statement.

4.17.1 Syntax

```
int altibase_stmt_param_count
(
    ALTIBASE_STMT stmt
)
```

4.17.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.17.3 Return Value

altibase_stmt_param_count() returns the number of parameter markers if successful, or ALTIBASE_INVALID_PARAMCOUNT if an error occurred.

4.17.4 Description

altibase_stmt_param_count() returns the number of parameter markers present in the prepared statement. If no parameter marker exists, the function returns 0. You must use this function after calling altibase_stmt_prepare().

4.17.5 Example

See the example in altibase_stmt_prepare().

4.18 altibase_stmt_prepare()

altibase_stmt_prepare() prepares the SQL statement and returns a status value.

4.18.1 Syntax

```
int altibase_stmt_prepare
(
    ALTIBASE_STMT  stmt,
    const char    *qstr
)
```

4.18.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle
const char *	qstr	Input	The SQL statement pointed to by the null-terminated string

4.18.3 Return Value

altibase_stmt_prepare() retunrs ALTIBASE_SUCCESS if the statement was successful. The function returns ALTIBASE_ERROR if an error occurred.

4.18.4 Description

altibase_stmt_prepare() prepares an SQL statement for execution. The SQL statement must be pointed to by null-terminated string. Normally the string must consist of a single SQL statement and you should not add a terminating semicolon(";"). Therefore, multiple-statement execution has not been enabled because the string cannot contain several statements separated by semicolons. To enable multiple-statement execution, you can process stored procedure that produce result sets.

The application can include one or more parameter markers in the SQL statement by embedding question mark ("?") characters into the SQL string at the appropriate positions. If embedding question mark ("?") characters into the SQL string, you must bind input data for the parameter markers by using altibase_stmt_bind_param() before calling altibase_stmt_execute().

The value for a statement attribute is available before you call altibase_stmt_prepare() to prepare other query, or altibase_stmt_close() or altibase_close() deallocates the statement handle. If you call one of these functions, a statement handle is reset. Therefore, the bind argument and a size of array are initialized. After this, if needing them, you must recreate them for a new query. You would realize the performance improvements with using altibase_query rather than altibase_stmt_prepare() and altibase_stmt_bind_param() because multiple-statement is enabled by calling altibase_stmt_execute() serveral times.

4.18.5 Example

```
rc = altibase_stmt_prepare(stmt, qstr);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    /* ... error handling ... */
}

printf("field count : %d\n", altibase_stmt_param_count(stmt));
printf("field count : %d\n", altibase_stmt_field_count(stmt));
```

4.19 altibase_stmt_get_attr()

altibase_stmt_get_attr() can be used to get the current value for a statement attribute.

4.19.1 Syntax

```
int altibase_stmt_get_attr
(
    ALTIBASE_STMT            stmt,
    ALTIBASE_STMT_ATTR_TYPE  option,
    void                    *arg
)
```

4.19.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	altibase	Input	Statment Handle
ALTIBASE_STMT_ATTR_TYPE	option	Input	The option argument is the option that you want to get.
void *	arg	Output	The arg is the output buffer.

4.19.3 Return Values

altibase_stmt_get_attr() retunrs ALTIBASE_SUCCESS if successful, or ALTIBASE_ERROR if an error occurred.

4.19.4 Description

You can get the current value for a statement attribute by calling altibase_stmt_get_attr(). You must allocate maximum size to buffer to store the value for a statement attribute because it is assumed that the size of the arg argument is large enough for buffer. For more details about the value for a statement attribute, see enum ALTIBASE_STMT_ATTR_TYPE.

4.19.5 Example

See the example in altibase_stmt_set_attr().

4.20 altibase_stmt_init()

altibase_stmt_init() creates an ALTIBASE_STMT handle.

4.20.1 Syntax

```
ALTIBASE_STMT altibase_stmt_init
(
    ALTIBASE altibase
)
```

4.20.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE	altibase	Input	Connection handle

4.20.3 Return Values

altibase_stmt_init() returns an ALTIBASE_STMT handle in case of success, or null if out of memory.

4.20.4 Description

altibase_stmt_init() creates an ALTIBASE_STMT handle with using a connection handle. An ALTIBASE_STMT handle should be freed with altibase_stmt_close() after you are done with the ALTIBASE_STMT handle.

4.20.5 Examples

```
stmt = altibase_stmt_init(altibase);
if (stmt == NULL)
{
    /* ... error handling ... */
}

/* ... omit ... */

rc = altibase_stmt_close(stmt);
if (! ALTIBASE_SUCCEEDE(rc))
{
    /* ... error handling ... */
}
```

4.21 altibase_stmt_processed()

altibase_stmt_processed() returns the number of rows after using the array binding.

4.21.1 Syntax

```
ALTIBASE_LONG altibase_stmt_processed
(
    ALTIBASE_STMT stmt
)
```

4.21.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.21.3 Return Value

altibase_stmt_processed() returns the number of rows after using the array binding in case of success, or ALTIBASE_INVALID_PROCESSED if an error occurred.

4.21.4 Description

altibase_stmt_processed() returns the number of rows after using the array binding. Normally, the value is same as a size of array. The function is used only with the array binding. You must not change or release its value as you please because it is managed within procedure.

4.21.5 Example

See 5.2 Array Binding in Chapter5.Using Array Binding and Array Fetching.

4.22 altibase_stmt_reset()

altibase_stmt_reset() resets statement handle for a prepared statement on client and server to state after prepare.

4.22.1 Syntax

```
int altibase_stmt_reset
(
    ALTIBASE_STMT stmt
)
```

4.22.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.22.3 Return Value

altibase_stmt_reset() returns ALTIBASE_SUCCESS if the statement was successful. The function returns ALTIBASE_ERROR if an error occurred.

4.22.4 Description

altibase_stmt_reset() resets statement handle for a prepared statement. The bind argument is the address of an array. The bind argument and a size of array are initialized. However, statement handle keeps prepared.

If a result set is returned before using the function, you must call altibase_stmt_free_result() first. Otehrwise, an error occurs.

4.23 altibase_stmt_result_metadata()

altibase_stmt_result_metadata() returns the result set metadata for the prepared query.

4.23.1 Syntax

```
ALTIBASE_RES altibase_stmt_result_metadata
(
    ALTIBASE_STMT stmt
)
```

4.23.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.23.3 Return Value

altibase_stmt_result_metadata() returns an ALTIBASE_RES containing meta information for the prepared query, or null if no meta information exists.

4.23.4 Description

altibase_stmt_result_metadata() returns the result set metadata for the prepared query such as SELECT statement that produces a result set. This result set pointer can be passed as an argument to any of the functions that process result set metadata, such as:

- altibase_field()
- altibase_fields()
- altibase_num_fields()

The result set should be freed when you are done with it, which you can do by passing it to altibase_free_result().

4.24 altibase_stmt_send_long_data()

This function is not enabled currently.

4.25 altibase_stmt_set_array_bind()

altibase_stmt_set_array_bind() specifies the size of array when you want to use the array binding.

4.25.1 Syntax

```
int altibase_stmt_set_array_bind
(
    ALTIBASE_STMT stmt,
    int           array_size
)
```

4.25.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle
int	array_size	Input	The size of array

4.25.3 Return Value

altibase_stmt_set_array_bind() returns ALTIBASE_SUCCESS if the statement was successful. The function returns ALTIBASE_ERROR if an error occurred.

4.25.4 Description

altibase_stmt_set_array_bind() specifies the size of array when you wnat to use the array binding. You can use the array binding only if its value is greater than 1. If canceling to use the array binding, you must set a size of array to 1.

The array biniding is valid only before calling altibase_stmt_reset(), altibase_stmt_prepare() and altibase_stmt_close(). altibase_stmt_set_array_bind() can be used after calling altibase_stmt_prepare() and before calling altibase_stmt_bind_param().

4.25.5 Example

See 5.2 Array Binding in Chapter5.Using Array Binding and Array Fetching.

4.26 altibase_stmt_set_array_fetch()

altibase_stmt_set_array_fetch() specifies the size of array when you want to fetch an array.

4.26.1 Syntax

```
int altibase_stmt_set_array_fetch
(
    ALTIBASE_STMT stmt,
    int           array_size
)
```

4.26.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle
int	array_size	Input	The size of array

4.26.3 Return Value

altibase_stmt_set_array_fetch() returns ALTIBASE_SUCCESS if the statement was successful. The function returns ALTIBASE_ERROR if an error occurred.

4.26.4 Description

altibase_stmt_set_array_fetch() specifies the size of array when you wnat to fetch an array. You can fetch an array only if the value returned by altibase_stmt_set_array_fetch() is greater than 1. If canceling to fetch an array, you must set a size of array to 1.

It is available to fetch an array only before calling altibase_stmt_reset(), altibase_stmt_prepare() and altibase_stmt_close(). altibase_stmt_set_array_bind() can be used after calling altibase_stmt_prepare() and before calling altibase_stmt_bind_result().

4.26.5 Example

See 5.3 Array Fetching in Chapter5.Using Array Binding and Array Fetching.

4.27 altibase_stmt_set_attr()

altibase_stmt_set_attr() can be used to affect behavior for a prepared statement. This function may be called to set the value for a statement attribute.

4.27.1 Syntax

```
int altibase_stmt_set_attr
(
    ALTIBASE_STMT             stmt,
    ALTIBASE_STMT_ATTR_TYPE  option,
    const void              *arg
)
```

4.27.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	Stmt	Input	Statement handle
ALTIBASE_STMT_ATTR_TYPE	option	Input	The option argument is the option that you want to set.
const void *	arg	Input	The arg argument is the value for the option. arg should point to a variable that is set to the desired attribute value.

4.27.3 Result Values

altibase_stmt_set_attr() retunrs ALTIBASE_SUCCESS if successful, or ALTIBASE_ERROR if an error occurred.

4.27.4 Description

You can get the current value for a statement attribute by calling altibase_stmt_get_attr(). You must allocate maximum size to buffer to store the value for a statement attribute because it is assumed that the size of the arg argument is large enough for buffer. For more details about the value for a statement attribute, see enum ALTIBASE_STMT_ATTR_TYPE in Chapter4.Prepared Statement Function Descriptions.

4.27.5 Examples

```
int atomic_array = ALTIBASE_ATOMIC_ARRAY_ON;

rc = altibase_stmt_set_attr(stmt, ALTIBASE_STMT_ATTR_ATOMIC_ARRAY,
                            &atomic_array);
```

```
/* ... check return value ... */

rc = altibase_stmt_get_attr(stmt, ALTIBASE_STMT_ATTR_ATOMIC_ARRAY,
                            &atomic_array);
/* ... check return value ... */

printf("Atomic array : %d\n", atomic_array);
```

4.28 altibase_stmt_status()

altibase_stmt_status() returns the results returned after uisng the array binding or fetching an array.

4.28.1 Syntax

```
unsigned short * altibase_stmt_status
(
    ALTIBASE_STMT stmt
)
```

4.28.2 Arguments

Data Type	Argument	In/Out	Description
ALTIBASE_STMT	stmt	Input	Statement handle

4.28.3 Return Value

altibase_stmt_status() returns an array from an Altibase result set of a query using the array binding or fetching an array in case of success.

4.28.4 Description

altibase_stmt_status() returns the results returned only after using the array binding or fetching an array depending on fetching an array already. If you do not fetch an array, the results are returned by using the array binding as follows.

State Value	Description
ALTIBASE_PARAM_SUCCESS	Execution was successful.
ALTIBASE_PARAM_SUCCESS_WITH_INFO	Execution was successful. However, error also occurred.
ALTIBASE_PARAM_ERROR	An error occurs when you use parameter.
ALTIBASE_PARAM_UNUSED	This parameter set is not enabled because the interruption occurs while you use the parameter set previously.

If you fetched an array already, the results are returned by fetching an array as follows.

State Value	Description
ALTIBASE_ROW_SUCCESS	Execution was successful.
ALTIBASE_ROW_SUCCESS_WITH_INFO	Execution was successful. However, error also occurred.
ALTIBASE_ROW_NOROW	This is an array element which is not fetched when the number of the fetched rows is lower than the size of array.
ALTIBASE_ROW_ERROR	An error occurred when you fetched a row.

You must not change or release the values as you please because they are managed within procedure.

4.28.5 Example

See Chapter5: Using Array Binding and Array Fetching.

4.28 altibase_stmt_status()

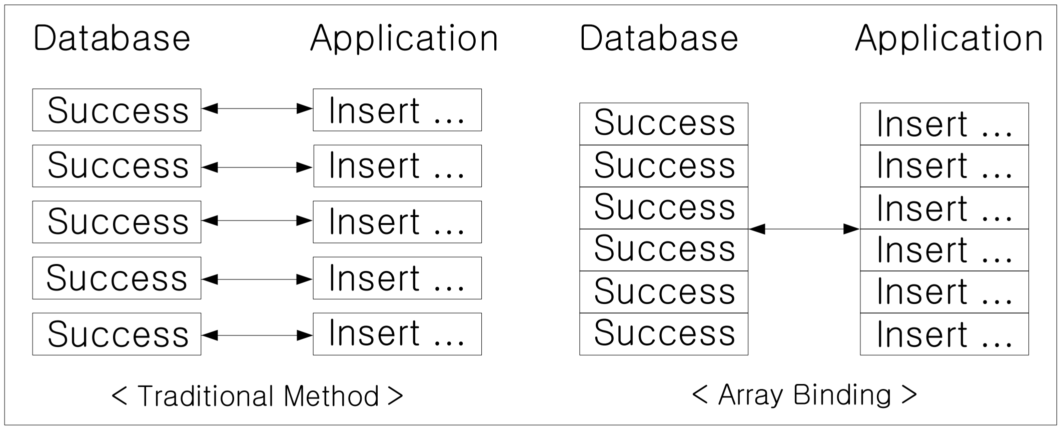
5 Using Array Binding and Array Fetching

This chapter covers array binding and array fetching to insert multiple data with one executing statement.

5.1 Overview

Using the array binding and array fetching allows you to pass the array of parameters with one executing statement. These methods can provide a decrease in network round-trip time and significant performance benefits when moving large amounts of data.

The following figure illustrates the array binding operation briefly. With using this, it takes shorter time for more data to be transfered because signals sent or received in networks decrease.



For more details about using the array binding and array fetching, see *ODBC Reference*.

5.1.1 How to Set ALTIBASE_BIND

To use the array binding and array fetching, you should use ALTIBASE_BIND as you whould when using traditional method. After this, you can set the bind argument which is the address of an array of ALTIBASE_BIND by using altibase_stmt_bind_param() or altibase_stmt_bind_result().

If specifying a size of an array by using altibase_stmt_set_array bind() or altibase_stmt_set_array_fetch(), ACI automatically recognizes that you want to use the array binding or array fetching, and considers that the bind argument is sent in suitable usage for the array binding and array fetching. Using the array binding or array fetching requires special care in specifying buffer and buffer_length.

This section includes following topics:

- buffer
- buffer_length

5.1.1.1 buffer

The buffer must be sufficiently large to hold the number of rows specified in array size. The folloiwng example shows how the application should allocate a large enough buffer if array size is set to 5 and column type is ALTIBASE_BIND_INTEGER in the buffer.

```
#define ARRAY_SIZE 5
```

```
/* ... omit ... */

int           int_dat[ARRAY_SIZE];
ALTIBASE_BIND bind;

/* ... omit ... */

bind.buffer_type   = ALTIBASE_BIND_INTEGER;
bind.buffer        = int_dat;
```

If the total length of the buffer is greater than array size, the rest is ignored except the number of rows specified in array size.

5.1.1.2 buffer_length

buffer_length must be specified as the fixed value if you want to use variable-length data such as ALTIBASE_BIND_STRING contained in the buffer. The following example shows how to set buffer_length if array size is set to 5 and maximum size of CHAR is specified as 50.

```
#define ARRAY_SIZE 5
#define STR_SIZE   50

/* ... omit ... */

int           str_dat[ARRAY_SIZE][STR_SIZE];
ALTIBASE_BIND bind;

/* ... omit ... */

bind.buffer_type   = ALTIBASE_BIND_STRING;
bind.buffer        = snt_dat;
bind.buffer_length = STR_SIZE;
```

If the buffer contains fixed-length data such as ALTIBASE_BIND_INTEGER, database ignores the data buffer length. Therefore, you need not set buffer_length for fixed-length data. If buffer_length is set to 0 for initialization, the buffer is assumed to have a large enough to hold the data.

5.2 Array Binding

The array binding allows you to pass the array of parameters with one executing statement. This method can provide significant performance benefits. The example of using this method is provided as follows.

5.2.1 Example

The struct ALTIBASE_BIND is used to define information for the binding operation. This data type contains the following members for use by application programs.

```
#define ARRAY_SIZE 2
#define PARAM_COUNT 2
#define STR_SIZE    50
#define QSTR        "INSERT INTO t1 VALUES (?, ?)"

int           int_dat[ARRAY_SIZE];
char          str_dat[ARRAY_SIZE][STR_SIZE];
ALTIBASE_LONG length[PARAM_COUNT][ARRAY_SIZE];

ALTIBASE      altibase;
ALTIBASE_STMT stmt;
ALTIBASE_BIND bind[PARAM_COUNT];
int           rc;
int           i;

/* ... omit ... */

int_dat[0] = 1;
int_dat[1] = 2;
strcpy(str_dat[0], "test1");
strcpy(str_dat[1], "test2");

length[0][0] = sizeof(int);
length[0][1] = sizeof(int);
length[1][0] = strlen(str_dat[0]);
length[1][1] = ALTIBASE_NTS;

memset(bind, 0, sizeof(bind));

bind[0].buffer_type   = ALTIBASE_BIND_INTEGER;
bind[0].buffer        = int_dat;
bind[0].length        = length[0];

bind[1].buffer_type   = ALTIBASE_BIND_STRING;
bind[1].buffer        = str_dat;
bind[1].buffer_length = STR_SIZE;
bind[1].length        = length[1];

stmt = altibase_stmt_init(altibase);
/* ... check return value ... */

rc = altibase_stmt_prepare(stmt, QSTR);
/* ... check return value ... */

rc = altibase_stmt_set_array_bind(stmt, ARRAY_SIZE);
/* ... check return value ... */

rc = altibase_stmt_bind_param(stmt, bind);
/* ... check return value ... */
```

```
rc = altibase_stmt_execute(stmt);
/* ... check return value ... */

printf("processed : %d\n", altibase_stmt_processed(stmt));
for (i = 0; i < ARRAY_SIZE; i++)
{
    printf("%d status : %d\n", , altibase_stmt_status(stmt)[i]);
}
```

5.3 Array Fetching

At fetch time, multiple rows worth of a column are copied to an array of variable by using the array fetching, This method can provide significant performance benefits.

When using the array fetching, you must use altibase_stmt_fetched() to check actual number of rows fetched shortly after fetching result as an array because the array size can be greater than the size of row data from the result set in a prepared statement by calling altibase_stmt_fetch(). If the array size is greater than the value fetched by altibase_stmt_fetched(), no more data exists. The example of using this method is provided as follows.

5.3.1 Example

The struct ALTIBASE_BIND is used to define information for the binding operation. This data type contains the following members for use by application programs.

```
#define ARRAY_SIZE 2
#define FIELD_COUNT 2
#define STR_SIZE    50
#define QSTR        "SELECT * FROM t1"

int           int_dat[ARRAY_SIZE];
char          str_dat[ARRAY_SIZE][STR_SIZE];
ALTIBASE_LONG length[FIELD_COUNT][ARRAY_SIZE];
ALTIBASE_BOOL is_null[FIELD_COUNT][ARRAY_SIZE];

ALTIBASE      altibase;
ALTIBASE_STMT stmt;
ALTIBASE_BIND bind[FIELD_COUNT];
int           rc;
int           i;
int           row;
int           fetched;
int           status;

/* ... omit ... */

stmt = altibase_stmt_init(altibase);/* ... check return value ... */

rc = altibase_stmt_prepare(stmt, QSTR);
/* ... check return value ... */

rc = altibase_stmt_execute(stmt);
/* ... check return value ... */

rc = altibase_stmt_set_array_fetch(stmt, ARRAY_SIZE);
/* ... check return value ... */

memset(bind, 0, sizeof(bind));

bind[0].buffer_type   = ALTIBASE_BIND_INTEGER;
bind[0].buffer        = int_dat;
bind[0].length        = length[0];
bind[0].is_null       = is_null[0];

bind[1].buffer_type   = ALTIBASE_BIND_STRING;
bind[1].buffer        = str_dat;
bind[1].buffer_length = STR_SIZE;
bind[1].length        = length[1];
bind[1].is_null       = is_null[1];
```

```
rc = altibase_stmt_bind_result(stmt, bind);
/* ... check return value ... */

do
{
    rc = altibase_stmt_fetch(stmt);
    if (rc == ALTIBASE_NO_DATA)
    {
        break;
    }
    if (ALTIBASE_NOT_SUCCEEDED(rc))
    {
        /* ... error handling ... */
        break;
    }

    fetched = altibase_stmt_fetched(stmt);
    for (i = 0; i < fetched; i++)
    {
        printf("row %d : ", row);
        status = altibase_stmt_status(stmt)[i];
        if (ALTIBASE_ROW_SUCCEEDED(status))
        {
            if (is_null[0][i] == ALTIBASE_TRUE)
            {
                printf("{null}");
            }
            else
            {
                printf("%d", int_dat[i]);
            }
            printf(", ");
            if (is_null[1][i] == ALTIBASE_TRUE)
            {
                printf("{null}");
            }
            else
            {
                printf("(%d) %s", length[1][i], str_dat[i]);
            }
        }
        else
        {
            printf("{status:%d}", status);
        }
        printf("\n");
        row++;
    }
} while (fetched == ARRAY_SIZE);
```

6 Using Failover

In this chapter, we introduce functions to use failover provided with Altibase. Failover is the capability to switch over automatically to a standby server upon abnormal termination of the previously active server.

6.1 How to Use Failover

The failover features is provided so that a fault that occurs while a database is providing service can continue to be provided as though no fault had occurred. For example, failover is the capability to switch over automatically to a standby server upon abnormal termination of the previously active server where you execute queries. This feature allows you to retry task which you wanted to do before abnormal termination without establishing a connection again in application. For more details, see *Replication Manual*.

This section includes the following topics:

- Failover-related Data Types
- How to Register a Failover Callback
- Task upon a Failover
- Example

6.1.1 Failover-related Data Types

This section discusses data types used for the failover and shows you how to use their values. The following topics are included:

- enum ALTIBASE_FAILOVER_EVENT
- altibase_failover_callback_func

6.1.1.1 enum ALTIBASE_FAILOVER_EVENT

FailoverEvent enumerated values are used to specify the state of the failover. If you register the failover callback function, the failover caliback function is notified of values returned by this data type. They are used when the failover callback function determines its advance to the next step

Table lists all the FailoverEvent enumeration values with a description of each enumerated value.

Enumerated Value	Description
ALTIBASE_FO_BEGIN	Indicates that failover has detected a lost connection and that failover is starting.
ALTIBASE_FO_END	Indicates successful completion of failover.
ALTIBASE_FO_ABORT	Indicates thata failover was unsuccessful, and there is no option of retrying.
ALTIBASE_FO_GO	FailOverCallback sends this so that STF (Service Time Failover) can advance to the next step.
ALTIBASE_FO_OUT	FailOverCallback sends this to prevent STF from advancing to the next step.

6.1.1.2 altibase_failover_callback_func

This is used to specify the function for failover callback.

```
typedef ALTIBASE_FAILOVER_EVENT (*altibase_failover_callback_func)
(
    ALTIBASE                 altibase,
    void                    *app_context,
    ALTIBASE_FAILOVER_EVENT  event
);
```

app_context argument is used to receive data for failover callback function. If you pass a pointer of data type used for registering callback to the function, the result is returned by app_context argument for callback function.

event argument is used to inform you that what kind of the failover event is raised. For more details, see enum ALTIBASE_FAILOVER_EVENT.

6.1.2 How to register a Failover Callback

You can define pointers to callback function and data needed for callback function. They are used when the failover occurs. You must register the callback for data to which you want the callback to apply after success in connection.

```
#define CONNSTR "DSN=127.0.0.1;PORT_NO=20300;UID=sys;PWD=manager;" \
                "AlternateServers=(192.168.3.54:20300,192.168.3.53:20300);" \
                "ConnectionRetryCount=3;ConnectionRetryDelay=5;" \
                "LoadBalance=on;SessionFailOver=on;"

/* ... omit ... */

if ((altibase = altibase_init()) == NULL)
{
    /* ... error handling ... */
}

rc = altibase_connect(altibase, CONNSTR);
/* ... check return value ... */

rc = altibase_set_failover_callback(altibase, my_callback_func, &my_context);
/* ... check return value ... */

/* ... omit ... */

rc = altibase_set_failover_callback(altibase, NULL, NULL);
/* ... check return value ... */
```

If the callback function is applied to an Altibase as a connection handle, Altibase will use the failover. The effect of registering the callback function will apply to iitself and entire ALTIBASE_STMTs depending on the connection handle. If you do not want the influence extend all over ALTIBASE_STMTs, you must make other Altibase separately for ALTIBASE_STMT.

6.1.3 Task upon a Failover

Altibase database can achieve automatic failover, and then re-executes the complete set of work that was attempted but not completed. To achieve a successful failover due to an error in the database session where database was executing a command, database can implement logic that chathces the errors during failover, and then retry and failed work. If there were no errors, database returns ALTIBASE_FAILOVER_SUCCESS.

If database fails to achieve a failover or cannot complete the work even after a failover, database also retruns an error code depending on this case. While you use altibase_stmt_execute() or altibase_stmt_fetch(), an error message can be returned by the failure of work such as preparing a SQL statement or binding data. At this time, you must check if how you set the failover and server works normally because database can achieve a successful failover but fails to complete its actual work.

6.1.4 Example

The following example shows how to use the failover.

```
rc = altibase_stmt_prepare(stmt, qstr);
/* ... check return value ... */

/* ... omit ... */

:retry

rc = altibase_stmt_execute(stmt);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    if (altibase_stmt_errno(stmt) == ALTIBASE_FAILOVER_SUCCESS)
    {
        /* Database re-executes a statement because it is automatically pre-
pared. */
        goto retry;
    }
    else
    {
        /* ... error handling ... */
    }
}

/* altibase_stmt_store_result() is optional */
rc = altibase_stmt_store_result(stmt);
if (ALTIBASE_NOT_SUCCEEDED(rc))
{
    if (altibase_stmt_errno(stmt) == ALTIBASE_FAILOVER_SUCCESS)
    {
        /* Database re-executes a statement because it is automatically pre-
pared. */
        goto retry;
    }
    else
    {
        /* ... error handling ... */
    }
}

while (1)
{
    rc = altibase_stmt_fetch(stmt)
    if (ALTIBASE_NOT_SUCCEEDED(rc))
    {
        if (altibase_stmt_errno(stmt) == ALTIBASE_FAILOVER_SUCCESS)
        {
            /* re-execute */
            goto retry;
        }
        else
        {
            /* ... error handling ... */
            break;
        }
    }

    /* TODO something */
}
```

Upon a failover due to an error in the database session where database was executing a prepared statement, database can immediately re-execute the work that was attempted but not completed because database automatically prepares a SQL statement and binds data again.

Index