Altibase® Application Development

JDBC User's Manual

Release 7.1 (July 5, 2017)

ALTIBASE

Altibase® Application Development JDBC User's Manual
Release 7.1

Altibase Corp.
10F, Daerung PostTower II,
306, Digital-ro, Guro-gu, Seoul 08378, Korea
Telephone: +82-2-2082-1000 Fax: 82-2-2082-1099
Homepage: http://www.altibase.com

Contents

Preface

About This Manual

This manual explains how to use the Altibase JDBC Driver.

Audience

This manual has been prepared for the following users of Altibase:

- Database administrators
- Application developers
- Programmers

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

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

Software Environment

This manual has been prepared assuming that Altibase 7.1 will be used as the database server.

Organization

This manual is organized as follows:

- Chapter1: Starting JDBC

This chapter offers basic instructions on how to use the Altibase JDBC driver.

- Chapter2: Basic Functions

This chapter explains how to connect to the database server with an IPv6 address and comparatively describes three statements that can be used in JDBC application programs.

- Chapter3: Advanced Functions

This chapter introduces advanced functions provided by the Altibase JDBC driver and explains how to use them.

- Chapter4: Tips & Recommendations

This chapter provides instructions for the efficient use of the Altibase JDBC driver.

- Chapter5: Error Messages

This chapter lists the SQL States of errors which can occur while using the Altibase JDBC driver.

- Appendix A: Data Type Mapping

This appendix lists the compatibility between Altibase data types and standard JDBC data types/Java data types.

Documentation Conventions

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

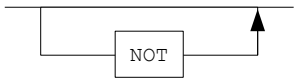
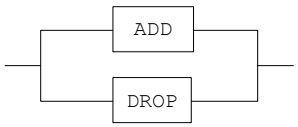
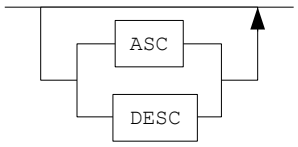
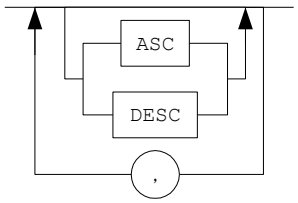
There are two sets of conventions:

- Syntax Diagram Conventions
- Sample Code Conventions

Syntax Diagrams

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

Elements	Meaning
	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.

Elements	Meaning
	Indicates a mandatory element.
	Indicates an optional element.
	Indicates a mandatory element comprised of options. One, and only one, option must be specified.
	Indicates an optional element comprised of options.
	Indicates an optional element in which multiple elements may be specified. A comma must precede all but the first element.

Sample Code Conventions

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

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

Convention	Meaning	Example
[ ]	Indicates an optional item.	VARCHAR [(size)] [[FIXED |] VARIABLE]
{ }	Indicates a mandatory field for which one or more items must be selected.	{ ENABLE | DISABLE | COMPILE }

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

Related Reading

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

- Administrator's Manual
- Replication Manual
- Spatial SQL Reference

Online Manuals

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

Altibase Welcomes Your Comments

Please feel free to send us your comments and suggestions regarding this manual. Your comments and suggestions are important to us, and may be used to improve future versions of the manual.

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

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

For immediate assistance with technical issues, please contact Altibase's Customer Support site (http://altibase.com/support-center/).

We always appreciate your comments and suggestions.

1 Starting JDBC

This chapter offers basic instructions on how to use the Altibase JDBC driver.

1.1 Installing the JDBC Driver

Download and install the Altibase package from the Altibase homepage (www.altibase.com).

Once the package is installed, the Altibase JDBC driver can be found in the $ALTIBASE_HOME/lib directory.

1.1.1 Version Compatibility

The Altibase 7.1 JDBC driver is a Type 4 pure Java JDBC driver, which conforms to the JDBC 3.0 specification and operates normally on JDK 1.5. or higher.

1.1.2 Checking the JDBC Driver Version

The versions of the installed JDBC driver and the JDK with which the driver has been compiled can be checked as below:

```
$ java -jar $ALTIBASE_HOME/lib/Altibase.jar
JDBC Driver Info : Altibase 7.1.0.0.0 with CMP 7.1.3 for JDBC 3.0 compiled with JDK 5
```

1.1.3 Setting the CLASSPATH

To use Altibase JDBC, the Altibase JDBC driver must be added to the CLASSPATH environment variable. Altibase provides both Altibase.jar file supporting logging function and Altibase_t.jar file which does not support the logging function.

```
e.g.) When using the bash shell in the Unix environment
export CLASSPATH=$ALTIBASE_HOME/lib/Altibase.jar:.:$CLASSPATH
```

1.2 Connecting to the Database

This section offers basic instructions through program codes on how to connect to the Altibase server with JDBC.

1.2.1 Loading the Driver

How to load the Driver class of the Altibase JDBC driver and locate the driver are shown below:

- Register the Altibase JDBC driver with the DriverManager.

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

- Acquire the driver from the DriverManager with the connection URL.

```
String sURL = "jdbc:Altibase://localhost:20300/mydb";
Driver sDriver = DriverManager.getDriver( sURL );
```

1.2.2 Configuring Connection Information

Multiple connection information can be passed to the driver with the Properties object. The following is a code example which sets connection attributes in the Properties object.

```
Properties sProps = new Properties();
sProps.put("user", "SYS");
sProps.put("password", "MANAGER");
```

1.2.3 Connecting to the Database

The database can be connected to with the connection URL and connection information. The following is a code example which connects to the database and acquires the Connection object.

```
Connection sCon = sDriver.connect(sURL, sProps);
```

The format of the connection URL used for the argument of the connect method above is shown below:

```
jdbc:Altibase://server_ip:server_port/dbname
```

1.2.4 Examples

```
import java.sql.Connection;
import java.sql.Driver;
import java.sql.DriverManager;
import java.util.Properties;
public class ConnectionExample
```

```
{
    public static void main(String[] args) throws Exception
    {
        String sURL = "jdbc:Altibase://localhost:20300/mydb";
        Properties sProps = new Properties();
        sProps.put("user", "SYS");
        sProps.put("password", "MANAGER");
        Class.forName("Altibase.jdbc.driver.AltibaseDriver");
        Driver sDriver = DriverManager.getDriver(sURL);
        Connection sCon = sDriver.connect(sURL, sProps);
    }
}
```

1.2.5 Compiling and Running

You can compile and run a JDBC application as follows:

```
$ javac ConnectionExample.java
$ java ConnectionExample
```

1.3 Connection Information

This section offers information of connection attributes available for use when connecting to Altibase, and explains how to set connection attributes.

1.3.1 Setting Connection Attributes

Connection attributes necessary for database connection can be set in the Properties object or specified in the connection URL.

1.3.1.1 Using the Connection URL

The value of the property can be specified by suffixing a "?" at the end of the connection URL and following it with the "key=value" format. Multiple properties can be input by connecting them with an "&".

The following is an example of a connection URL.

```
"jdbc:Altibase://localhost:20300/mydb?fetch_enough=0&time_zone=DB_TZ"
```

1.3.1.2 Using the Properties Object

After the Properties object has been created and the key and value have been input, it is ready to be used as connection information.

The following example uses the Properties object:

```
Properties sProps = new Properties();
sProps.put("fetch_enough", "30");
sProps.put("time_zone", "DB_TZ");

...

Connection sCon = DriverManager.getConnection( sURL, sProps );
```

1.3.2 About Connection Attributes

This section offers descriptions of the connection attributes available for use when connecting to Altibase. The following items are included in the description of each attribute:

- Default value: The value used by default, if no other value is specified
- Range: The value available for specification
- Mandatory: Whether or not the attribute must be specified

- Setting range: Whether the attribute which is set affects the whole system or only affects the given session.
- Description

1.3.2.1 alternateservers

Default Value	
Range	[host_name:port_number[/dbname] [, host_name:port_number[/dbname]]*
Mandatory	No
Setting Range	
Description	The list of servers available for connection in the event of a Connection Failover. For instructions on its use, please refer to “JDBC and Failover” in Chapter 3.

1.3.2.2 app_info

Default Value	
Range	A random string
Mandatory	No
Setting Range	The session
Description	Specifies the string to be stored in the CLIENT_APP_INFO column in V$SESSION.

1.3.2.3 auto_commit

Default Value	true
Range	[true | false]
Mandatory	No
Setting Range	The session

Description	Sets whether or not to automatically commit the transaction when the execution of a statement is complete.

1.3.2.4 defer_prepares

Default Value	off
Range	[on | off]
Mandatory	No
Setting Range	The session
Description	This can specify whether or not to hold the communication with server when PrepareStatement is called. If only one connection is shared by multiple threads, it will not work properly. If this attribute is set to ON, prepare request is not sent to the server until the Execute function is called even if the PrepareStatement is called. However, the prepare request is immediately sent to the server if the following methods listed below are called after PrepareStatement (). getMetData getParameterMetaData setObject(int, Object, int)

1.3.2.5 ciphersuite_list

Default Value	Please refer to the list of supported cipher suites for JRE.
Range	Random string
Mandatory	No
Setting Range	N/A
Description	This is a list of available ciphers for communication through SSL.

1.3.2.6 clientside_auto_commit

Default Value	off
Range	[on | off]
Mandatory	No

Setting Range	The session
Description	Sets whether autocommit operations are to be controlled by the Altibase server or the JDBC driver. The value set for this attribute is applied only if the auto_commit attribute is set to true or omitted. on: JDBC driver controls autocommit. off: Altibase server controls autocommit. For further information, please refer to “Controlling Autocommit” in Chapter 3.

1.3.2.7 connectionretrycount

Default Value	0
Range	A numerical value within the Unsigned Integer range [1 - 2^{31}]
Mandatory	No
Setting Range	
Description	Specifies the number of times reconnection is attempted to another server in the event of a Connection Failover. If this value is 1, reconnection to another server is attempted once; therefore, connection is attempted two times in total. For instructions on its use, please refer to “JDBC and Failover” in Chapter 3.

1.3.2.8 connectionretrydelay

Default Value	0
Range	A numerical value within the Unsigned Integer range [1 - 2^{31}]
Mandatory	No
Setting Range	
Description	Specifies the waiting time during which connection is attempted to another server when the ConnectionRetryCount is set in the event of a Connection Failover. The unit is seconds. For instructions on its use, please refer to “JDBC and Failover” in Chapter 3.

1.3.2.9 database

Default Value	mydb
Range	The database name
Mandatory	No
Setting Range	N/A
Description	The name of the database created in the Altibase server to be connected

1.3.2.10 datasource

Default Value	
Range	[0 - 65535]
Mandatory	Mandatory to use the Datasource
Setting Range	N/A
Description	The name of the DataSource

1.3.2.11 date_format

Default Value	Sets the value of the ALTIBASE_DATE_FORMAT environment variable. On omission, the in/output format of the DATE type takes the value set for the server.
Range	A DATE format string
Mandatory	No
Setting Range	The system
Description	Sets the in/output format of the DATE type. If the format is set and a data of another format is input by the client, instead of treating it as a DATE type, an error is returned.

1.3.2.12 description

Default Value	
Range	A random string
Mandatory	No
Setting Range	N/A
Description	The description part of the DataSource

1.3.2.13 ddl_timeout

Default Value	0
Range	A numerical value within the Unsigned Integer range
Mandatory	No
Setting Range	The session
Description	Sets the time limit for the execution of DDL statements. Queries which exceed the execution time limit are automatically terminated. The unit is seconds. The value 0 indicates infinity.

1.3.2.14 fetch_enough

Default Value	0
Range	[0 - 2147483647]
Mandatory	No
Setting Range	The session
Description	Sets the FetchSize of the current session. This indicates the number of rows to be retrieved per FETCH from the server. If this value is 0, the JDBC driver fetches the maximum size of data that can be contained in one network packet from the server.

1.3.2.15 fetch_timeout

Default Value	60
Range	A numerical value within the Unsigned Integer range
Mandatory	No
Setting Range	The session
Description	Sets the time limit for the execution of SELECT statements. Queries which exceed the execution time limit are automatically terminated. The unit is seconds. The value 0 indicates infinity.

1.3.2.16 idle_timeout

Default Value	0
Range	A numerical value within the Unsigned Integer range
Mandatory	No
Setting Range	The session
Description	Sets the time limit during which the Connection stays connected without performing any operations. The connection is automatically released when the time is up. The unit is seconds. The value 0 indicates infinity.

1.3.2.17 isolation_level

Default Value	
Range	[2 | 4 | 8]
Mandatory	No
Setting Range	The system
Description	2: TRANSACTION_READ_COMMITTED 4: TRANSACTION_REPEATABLE_READ 8: TRANSACTION_SERIALIZABLE

1.3.2.18 keystore_password

Default Value	N/A
Range	Random String
Mandatory	No
Setting Range	N/A
Description	Specifies the password for keystore_url.

1.3.2.19 keystore_type

Default Value	JKS
Range	[JKS, JCEKS, PKCS12, etc.]
Mandatory	No
Setting Range	N/A
Description	Specifies the keystore type for keystore_url.

1.3.2.20 keystore_url

Default Value	N/A
Range	Random String
Mandatory	No
Setting Range	N/A
Description	Specifies the path to the keystore (a container for its own private key and the certificates with their corresponding public keys).

1.3.2.21 lob_cache_threshold

Default Value	8192
Range	[0 - 8192]

Mandatory	No
Setting Range	The session
Description	Sets the maximum size of the LOB data to be cached on the client.

1.3.2.22 login_timeout

Default Value	
Range	A numerical value within the Unsigned Integer range
Mandatory	No
Setting Range	The session
Description	Sets the maximum waiting time for logging in. For further information, please refer to “Timeout” in Chapter 3.

1.3.2.23 max_statements_per_session

Default Value	
Range	A numerical value within the Signed Short range
Mandatory	No
Setting Range	The session
Description	Specifies the maximum number of statements available for execution in one session. The value 0 indicates infinity.

1.3.2.24 ncharliteralreplace

Default Value	false
Range	[true | false]
Mandatory	No
Setting Range	The session

Description	Specifies whether or not to check the existence of an NCHAR string(literal) within SQL statements on the client.

1.3.2.25 password

Default Value	
Range	
Mandatory	Yes
Setting Range	N/A
Description	The password of the user ID

1.3.2.26 port

Default Value	20300
Range	[0 - 65535]
Mandatory	No
Setting Range	N/A
Description	Specifies the port number of the server to be connected to. If ssl_enable is false, 20300 is used; otherwise, 20443 is used.

1.3.2.27 prefer_ipv6

Default Value	false
Range	[true | false]
Mandatory	No
Setting Range	
Description	Specifies whether to use the IPv6 address as it is, or to convert it to IPv4 for use. For further information, please refer to “IPv6 Connectivity”.

1.3.2.28 privilege

Default Value	
Range	[normal | sysdba]
Mandatory	No
Setting Range	N/A
Description	The connection mode normal: normal mode sysdba: DBA mode

1.3.2.29 query_timeout

Default Value	600
Range	A numerical value within the Unsigned Integer range
Mandatory	No
Setting Range	The session
Description	Sets the time limit for query execution. Queries which exceed the execution time limit are automatically terminated. The unit is seconds. The value 0 indicates infinity.

1.3.2.30 remove_redundant_transmission

Default Value	0
Range	[0 | 1]
Mandatory	No
Setting Range	The session
Description	Sets whether or not to use the duplicate data compression method for CHAR, VARCHAR, NCHAR, NVARCHAR type strings.

1.3.2.31 response_timeout

Default Value	
Range	A numerical value within the Unsigned Integer range
Mandatory	No
Setting Range	The session
Description	Sets the maximum waiting time for a response. For further information, please refer to “Timeout” in Chapter 3.

1.3.2.32 sessionfailover

Default Value	off
Range	[on | off]
Mandatory	No
Setting Range	
Description	Sets whether or not to use STF (Session Time Failover). For instructions on its use, please refer to “JDBC and Failover” in Chapter 3.

1.3.2.33 server

Default Value	localhost
Range	Please refer to “IPv6 Connectivity” in Chapter 2. Basic Functions of this manual.
Mandatory	Yes
Setting Range	N/A
Description	The IP address or host name of the Altibase server to be connected

1.3.2.34 ssl_enable

Default Value	false

Range	[true | false]
Mandatory	No
Setting Range	Session
Description	Specifies whether or not to connect to the database over SSL connection. For further information, please refer to the *Altibase SSL/TLS User's Guide*.

1.3.2.35 time_zone

Default Value	DB_TZ (The timezone set in the database is used)
Range	
Mandatory	No
Setting Range	The session
Description	Sets the time zone. For further information, please refer to the TIME_ZONE property in *General Reference*.

1.3.2.36 truststore_password

Default Value	N/A
Range	Random string
Mandatory	No
Setting Range	N/A
Description	Specifies the password for truststore_url.

1.3.2.37 truststore_type

Default Value	JKS
Range	[JKS, JCEKS, PKCS12, etc.]

Mandatory	No
Setting Range	N/A
Description	Specifies the truststore type for truststore_url.

1.3.2.38 truststore_url

Default Value	N/A
Range	Random string
Mandatory	No
Setting Range	N/A
Description	Specifies the path to the truststore (a keystore containing certificates that belong to the communication partners).

1.3.2.39 user

Default Value	
Range	
Mandatory	Yes
Setting Range	N/A
Description	The user ID of the database to be connected.

1.3.2.40 utrans_timeout

Default Value	3600
Range	A numerical value within the Unsigned Integer range
Mandatory	No
Setting Range	The session

Description	Sets the time limit for the execution of UPDATE statements. Queries which exceed the execution time limit are automatically terminated. The unit is seconds. The value 0 indicates infinity.

1.3.2.41 verify_server_certificate

Default Value	true
Range	[true | false]
Mandatory	No
Setting Range	N/A
Description	Specifies whether or not to authenticate the server's CA certificate. If this value is false, the client application will not authenticate the server's CA certificate.

1.4 Using Statement and ResultSet

This section offers basic instructions through program codes on how to connect to the Altibase server and execute SQL statements with JDBC. For convenience, instructions on how to handle exceptions are omitted.

1.4.1 Examples

```
import java.util.Properties;
import java.sql.*;

//...

String sURL      = "jdbc:Altibase://localhost:20300/mydb";
String sUser     = "SYS";
String sPassword = "MANAGER";

Connection sCon = null;

//Set properties for Connection
Properties sProps = new Properties();
sProps.put( "user",     sUser);
sProps.put( "password", sPassword);

// Load class to register Driver into DriverManager
Class.forName("Altibase.jdbc.driver.AltibaseDriver");

// Create a Connection Object
sCon = DriverManager.getConnection( sURL, sProps );

// Create a Statement Object
Statement sStmt = sCon.createStatement();

// Execute DDL Type Query
sStmt.execute("CREATE TABLE TEST ( C1 VARCHAR (10) )");

// Execute Inserting Query
sStmt.execute("INSERT INTO TEST VALUES ('ABCDE')");

// Execute Selecting Query
// Get Result Set from the Statement Object
ResultSet sRs = sStmt.executeQuery("SELECT * FROM TEST");

// Get ResultSetMetaData Object
ResultSetMetaData sRsMd = sRs.getMetaData();

// Retrieve ResultSet
while(sRs.next())
{
    for(int i=1; i<=sRsMd.getColumnCount(); i++)
    {
        // Get Actual Data and Printout
        System.out.println(sRs.getObject(i));
    }
}

// Eliminate ResultSet Resource
sRs.close();
```

```
// Execute Updating Query
sStmt.execute("UPDATE TEST SET C1 = 'abcde'");

// Execute Selecting Query
// Get Result Set from the Statement Object
sRs = sStmt.executeQuery("SELECT * FROM TEST");

// Get ResultSetMetaData Object
sRsMd = sRs.getMetaData();

// Retrieve ResultSet
while(sRs.next())
{
	for(int i=1; i<=sRsMd.getColumnCount(); i++)
	{
		// Get Actual Data and Printout
		System.out.println(sRs.getObject(i));
	}
}

// Eliminate ResultSet Resource
sRs.close();

// Execute Deleting Query
sStmt.execute("DELETE FROM TEST");

// Execute Selecting Query
// Get Result Set from the Statement Object
sRs = sStmt.executeQuery("SELECT * FROM TEST");

// Get ResultSetMetaData Object
sRsMd = sRs.getMetaData();

// Retrieve ResultSet
while(sRs.next())
{
	for(int i=1; i<=sRsMd.getColumnCount(); i++)
	{
		// Get Actual Data and Printout
		System.out.println(sRs.getObject(i));
	}
}

// Eliminate Resources
sRs.close();
sStmt.close();
```

1.5 JDBC Connection Failover

Due to the termination of one server in an environment where multiple Altibase servers are running, network failure or etc., the service of an application implemented with the Altibase JDBC driver can be compromised.

In the event of such a failure, the client which connected to the server on which the failure occurred detects the situation and automatically connects to another server and processes the statements that were being executed; this process is called a Fail-Over.

For instructions on how to use the Fail-Over feature in JDBC applications, please refer to Chapter 4. of *Replication Manual*.

2 Basic Functions

Database objects can be used with the Altibase JDBC driver in the same manner as using the standard JDBC interface.

This chapter explains how to connect to the database server with an IPv6 address and comparatively describes three statements that can be used in JDBC application programs.

2.1 IPv6 Connectivity

The Altibase JDBC driver supports the use of IPv6 addresses and host names conversible to IPv6 addresses in the JDBC URL.

2.1.1 Overview

To specify an IPv6 address in the URL, the address must be enclosed in square brackets ("[ ]"). For example, when specifying the localhost to an IP address, brackets are not used for writing the IPv4 address, 127.0.0.1. Meanwhile, brackets are necessary for using the IPv6 address, [::1]. For further information on IPv6 address notation, please refer to *Administrator's Manual*.

2.1.2 Prerequisites

2.1.2.1 java.net.preferIPv4Stack

To connect with an IPv6 address, the java.net.preferIPv4Stack property must be set to FALSE when the client runs.

If this property is set to TRUE, the client application cannot connect to the database server with an IPv6 address.

```
$ java -Djava.net.preferIPv4Stack=false sample [::1]
```

2.1.2.2 java.net.preferIPv6Addresses

Regardless of whether the java.net.preferIPv6Addresses property is set to TRUE or FALSE, the Altibase JDBC driver is not affected.

2.1.2.3 PREFER_IPV6

If the host name is input to the server_ip property of the URL, the JDBC driver converts the host name to an IPv4 address or an IPv6 address, depending on the value specified for the PREFER_IPV6 property.

If this property is TRUE, and the host name is input to the server_ip property, the client application first converts the host name to an IPv6 address.

If this property is omitted or set to FALSE, however, the client application program first converts the host name to an IPv4 address.

If the client application fails on its first attempt to connect, it will attempt to re-connect with an IP address of a different version from the first.

2.1.3 How to Use IPv6

For IP addresses of the IPv6 format, the address value is available for specification as it is.

Whether the JDBC driver is to convert the host name to an IPv4 address or an IPv6 address can be specified in the PREFER_IPV6 property.

```
Properties sProps = new Properties();
...
sProps.put( "PREFER_IPV6", "FALSE");
```

If the PREFER_IPV6 property is set to FALSE as above, the JDBC driver converts the host name to an IPv4 address. If the PREFER_IPV6 property is TRUE and the host name is given, the application first converts the host name to an IPv6 address.

If this property is omitted or set to FALSE, the client application first converts the host name to an IPv4 address. The basic operation of a JDBC driver is the conversion of the host name to an IPv4 address.

If the client application fails on its first attempt to connect with an IP address of a preferred version, it will attempt to re-connect with an IP address of another version.

2.1.4 Examples

```
Connection sCon = null;
Properties sProps = new Properties();

sProps.put( "user", "SYS");
sProps.put( "password", "MANAGER");
sProps.put( "PREFER_IPV6", "FALSE");

String sURL = “jdbc:Altibase://localhost:20300/mydb";
Connection sCon = DriverManager.getConnection( sURL, sProps );
```

2.2 Statement, PreparedStatement and CallableStatement

Depending on whether or not an in/out parameter is used in a SQL statement or whether or not a SQL statement is directly executed, different Statement objects are available for use in JDBC. The following table shows whether or not the PREPARE function and in/output parameters are available for use for each Statement.

	PREPARE	IN Parameter	OUT Parameter
Statement	X	X	X
PreparedStatement	O	O	X
CallableStatement	O	O	O

2.2.1 Statement

Statement is mainly used for directly executing static SQL statements.

2.2.2 PreparedStatement

PreparedStatement is mainly used for preparing the SQL statement before executing it. When the same statement is repeatedly executed, improved performance can be anticipated with the use of PreparedStatement, instead of Statement.

When the PreparedStatement object is created, the Altibase JDBC driver commands the server to PREPARE the statement. If the server fails to PREPARE, an error is returned and the JDBC driver throws an exception.

Unlike Statement, input parameters can be used for PreparedStatement. A parameter is indicated as the "?" character within a SQL statement and can be set to a value with the setXXX() method.

2.2.2.1 Examples

The following is a code example using IN parameters with PreparedStatement.

```
PreparedStatement sPrepStmt = sConn.prepareCall("INSERT INTO t1 VALUES (?, ?)");
sPrepStmt.setInt(1, 1);
sPrepStmt.setString(2, "string-value");
sPrepStmt.execute();
sPrepStmt.close();
```

2.2.3 CallableStatement

CallableStatement can be used with an input or output parameter. CallableStatement is mainly used for calling a stored procedure or a stored function.

2.2.3.1 Examples

The following is a code example using an IN parameter and an OUT parameter with CallableStatement.

```
CallableStatement sCallStmt = connection().prepareCall("{call p1(?, ?)");
sCallStmt.setInt(1, 1);
sCallStmt.registerOutParameter(2, Types.VARCHAR);
sCallStmt.execute();

String sOutVal = sCallStmt.getString(2);
// todo something ...

sCallStmt.close();
```

2.3 Using the National Character Set

This section offers instructions on how to use national character strings on UNICODE types, such as NCHAR or NVARCHAR types, in JDBC.

2.3.1 Retrieving and Altering Data

Data of NCHAR/NVARCHAR data types can be retrieved and altered with JDBC in the same manner as retrieving and altering data of CHAR/VARCHAR types. The same methods used for the CHAR data type, such as getString and setString etc., can be used.

2.3.2 Using Constant Strings

How to use a constant string with a national character in a SQL statement is shown below:

- Set the NcharLiteralReplace property to TRUE when connecting to the server.
- To use a national character string in a SQL statement as a constant string, prefix 'N' to the string.

2.3.2.1 Example

```
// create table t1 (c1 nvarchar(1000));
Properties sProps;
sProps.put( "user", "SYS");
sProps.put( "password", "MANAGER");
sProps.put( "NcharLiteralReplace", "true");
Connection sCon = DrierManager.getConnection( sURL, sProps );

Statement sStmt = sCon.createStatement();
sStmt.execute("insert into t1 values (N'AB 가나')");
ResultSet sRS = sStmt.executeQuery( "select * from t1 where c1 like N'%가나%'");
```

3 Advanced Functions

This chapter introduces advanced functions provided by the Altibase JDBC driver and explains how to use them.

3.1 Auto-generated Keys

Auto-generated keys are values which distinctly point to each row in a table, and are automatically generated in the database.

In Altibase, a sequence can act as auto-generated keys. This section explains how to obtain the values of auto-generated keys in JDBC.

3.1.1 How to Use Auto-generated Keys

To obtain an auto-generated key, first the Statement object is executed with a method specifying the column for which auto-generated keys are to be obtained. The ResultSet of the auto-generated keys can be retrieved with the getGeneratedKeys() method.

Or, after having created the PreparedStatement object with a method specifying the column for which auto-generated keys are to be obtained and executing it, the ResultSet of the auto-generated keys can be retrieved with the getGeneratedKeys() method.

The following are Statement methods that execute SQL statements which retrieve auto-generated keys.

```
public boolean execute(String aSql, int aAutoGeneratedKeys) throws SQLException;
public boolean execute(String aSql, int[] aColumnIndexes) throws SQLException;
public boolean execute(String aSql, String[] aColumnNames) throws SQLException;
```

The following are Connection methods that create the PreparedStatement object which retrieve auto-generated keys.

```
public PreparedStatement prepareStatement(String aSql, int aAutoGeneratedKeys) throws SQLException;
public PreparedStatement prepareStatement(String aSql, int[] aColumnIndexes) throws SQLException;
public PreparedStatement prepareStatement(String aSql, String[] aColumnNames) throws SQLException;
```

After having executed a SQL statement in one of the above two ways, auto-generated keys can be obtained by a ResultSet object with the following Statement method.

```
public ResultSet getGeneratedKeys() throws SQLException;
```

3.1.2 Restrictions

When obtaining auto-generated keys in Altibase, the following restrictions apply:

- Its use is only supported for simple INSERT statements.

- Since Altibase does not support columns with the AUTO INCREMENT property, auto-generated keys can only be obtained from a sequence.

The following is an example of a SQL statement from which auto-generated keys can be obtained.

```
INSERT INTO t1 (id, val) VALUES (t1_id_seq.nextval, ?);
```

The following is an example of a SQL statement from which auto-generated keys cannot be obtained.

```
SELECT * FROM t1;
EXEC p1;
```

If a SQL statement which does not produce auto-generated keys is executed with the generator flag (Statement.RETURN_GENERATED_KEYS), the flag is ignored and the getGeneratedKeys() method returns an empty result set.

3.1.3 Examples

```
sStmt.executeUpdate(sQstr, Statement.RETURN_GENERATED_KEYS);
ResultSet sKeys = sStmt.getGeneratedKeys();
while (sKeys.next())
{
      int sKey = sKeys.getInt(1);

      // do somethings...
}
sKeys.close();
sStmt.close();
```

3.2 Timeout

This section gives an explanation of timeouts which can occur in a client session connected to the Altibase server and provides code examples to show how to set properties related to timeouts.

3.2.1 Login Timeout

A login timeout occurs when a connect method of a Connection object is called and a response is not received from the server within the maximum waiting time. The maximum waiting time is set in the login_timeout property and the unit is seconds.

3.2.1.1 Code Examples

The following are code examples which show two ways to set the login_timeout property.

1. Create a Connection object with the Properties object to which the timeout property has been added.

```
Properties sProps = new Properties();
...
sProps("login_timeout", "100");
...
Connection sCon = DriverManager.getConnection( sUrl, sProps );
```

2. Create a Connection object with a connection URL which specifies the timeout property.

```
String sUrl = "jdbc:Altibase://localhost:20300/mydb?login_timeout=100";
Connection sCon = DriverManager.getConnection( sUrl );
```

3.2.2 Response Timeout

A response timeout occurs when the maximum waiting time for a response from the Altibase server is exceeded. The maximum waiting time is set in the response_timeout property and the unit is seconds.

This value is applied to all methods which communicate with the server.

3.2.2.1 Code Examples

The following are code examples which show different ways to set the response_timeout property.

1. Create a Connection object with a Properties object to which the timeout property has

been added.

```
Properties sProps = new Properties();
...
sProps("response_timeout", "100");
...
Connection sCon = DriverManager.getConnection( sUrl, sProps );
```

2. Create a Connection object with a connection URL which specifies the timeout property.

```
String sUrl = "jdbc:Altibase://localhost:20300/mydb?response_timeout=100";
Connection sCon = DriverManager.getConnection( sUrl );
```

3. Pass it as an argument when the application is running.

```
java ... -DALTIBASE_RESPONSE_TIMEOUT=100 ...
```

4. Set the environment variable.

```
// Linux
export ALTIBASE_RESPONSE_TIMEOUT=100
```

3.3 DataSource

The Altibase JDBC driver offers a way to connect to the database with a file that contains connection configurations. Datasource is the set of connection information to a database server in the configuration file.

3.3.1 How to set DataSource

DataSource is set in the following format to the altibase_cli.ini file.

```
# comment

[ datasource_name ]
Server=localhost # comment
Port=20300
User=sys
Password=manager
```

Additional connection properties can be added by writing strings of the "key=value" format in lines.

The JDBC driver searches for the altibase_cli.ini file in the paths of the following order:

1. /altibase_cli.ini
2. $HOME/altibase_cli.ini
3. $ALTIBASE_HOME/conf/altibase_cli.ini

3.3.2 Connecting with DataSource

To connect to the server with DataSource, the DSN (DataSouce Name) specified in the altibase_cli.ini file should be specified in a connection URL, instead of the IP address and port number.

The following is an example of a connection URL using a DSN.

```
jdbc:Altibase://datasource_name
jdbc:Altibase://datasource_name:20301
jdbc:Altibase://datasource_name:20301?sys=user&password=pwd
```

When specifying a DSN in a connection URL, port or other properties can be additionally specified. If a property specified in the altibase_cli.ini file is duplicately specified in a connection URL, however, the file value is ignored and the connection URL value is used.

3.4 Connection Pool

A Connection Pool can be set and managed in the following manner.

- Use AltibaseConnectionPoolDataSource: When using a Connection Pool in WAS, specify this class in the JDBC Connection Pool configuration of WAS. The name of this class was ABConnectionPoolDataSource for Altibase JDBC drivers of versions prior to 7.1

3.4.1 Configuring WAS (Web Application Server)

Altibase can be used with the following web application servers.

- Tomcat 8.x
- WebLogic 12.x
- Jeus 6.x

For further information on how to configure and use the connection pool and JDBC driver on each web application, please refer to their manuals.

3.4.1.1 Tomcat 8.x

For further information on how to install and configure Apache Tomcat, please refer to http://tomcat.apache.org/tomcat-8.0-doc/index.html.

Context configuration

Add the JNDI DataSource to the Context as below:

```
<Context>


  <Resource name="jdbc/altihdb" auth="Container" type="javax.sql.DataSource"
 driverClassName="Altibase.jdbc.driver.AltibaseDriver"
                  url="jdbc:Altibase://localhost:20300/mydb" username="SYS" password="MANAGER"
maxTotal="100" maxIdle="30" maxWaitMillis="10000" />

</Context>
```

web.xml configuration

```
<!----- web.xml ----->
<resource-ref>
  <description>Altibase Datasource example</description>
  <res-ref-name>jdbc/altihdb</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Code example

```
Context initContext = new InitialContext();
Context envContext    = (Context)initContext.lookup("java:/comp/env");
DataSource ds = (DataSource)envContext.lookup("jdbc/altihdb");
Connection conn = ds.getConnection();
// ...
```

3.4.1.2 WebLogic 12.x

For further information on how to install and configure the WebLogic server, please refer to http://docs.oracle.com/middleware/1213/wls/index.html.

You can configure the connection pool and the JDBC datasource by referring to the following links:

- http://docs.oracle.com/middleware/1213/wls/WLACH/taskhelp/jdbc/jdbc_datasources/CreateDataSources.html
- http://docs.oracle.com/middleware/1213/wls/WLACH/pagehelp/JDBCjdbcdatasourcesjdbcdatasourceconfigconnectionpooltitle.html

The configuration options for Altibase are as follows:

- Database Type: Other
- Driver Class Name: Altibase.jdbc.driver.AltibaseDriver
- URL: jdbc:Altibase://localhost:20300/mydb

3.4.1.3 Jeus 6.x

Set the Connection Pool by editing the <DataSource> element of the $JEUS_HOME/config/JeusMain.xml file.

```
<!------ JeusMain.xml ----->
<resource>
    <data-source>
        <database>
            <vendor>others</vendor>
            <export-name>jdbc/altihdb</export-name>
            <data-source-class-name>
                Altibase.jdbc.driver.AltibaseConnectionPoolDataSource
            </data-source-class-name>
            <data-source-type>ConnectionPoolDataSource</data-source-type>
            <auto-commit>true</auto-commit>
            <property>
                <name>PortNumber</name>
                <type>java.lang.Integer</type>
                <value>20300</value>
            </property>
            <property>
                <name>Password</name>
                <type>java.lang.String</type>
                <value>MANAGER</value>
            </property>
```

```
                <property>
                    <name>ServerName</name>
                    <type>java.lang.String</type>
                    <value>localhost</value>
                </property>
                <property>
                    <name>ConnectionAttributes</name>
                    <type>java.lang.String</type>
                    <value>;create=true</value>
                </property>
                <property>
                    <name>DatabaseName</name>
                    <type>java.lang.String</type>
                    <value>mydb</value>
                </property>
                <property>
                    <name>User</name>
                    <type>java.lang.String</type>
                    <value>SYS</value>
                </property>
            </database>
        </data-source>
</resource>
```

3.5 Multiple ResultSet

PSM(Stored procedures and stored functions) for Altibase can return a multiple number of result sets to the client. Using an example that returns multiple result sets, this section offers instructions through a code example on how to use these result sets in JDBC applications.

The following is an example of a PSM which returns multiple result sets.

```
CREATE TYPESET my_type
AS
    TYPE my_cur IS REF CURSOR;
END;

CREATE PROCEDURE p1 (p1 OUT MY_TYPE.MY_CUR, p2 out MY_TYPE.MY_CUR)
AS
BEGIN
    OPEN p1 FOR 'SELECT * FROM t1';
    OPEN p1 FOR 'SELECT * FROM t2';
END;
```

The following is an example of a code which uses multiple result sets returned by a call to a PSM in a JDBC application.

```
CallableStatement sCallStmt = connection().prepareCall("{call p1()}");
sCallStmt.execute();
ResultSet sRs = null;
ResultSetMetaData sRsMd = null;

do{
    sRs = sCallStmt.getResultSet();
    sRsMd = sRs.getMetaData();

    if(sRsMd != null)
    {
        while(sRs.next())
        {
            // do something
            for(int i=1; i <= sRsMd.getColumnCount(); i++)
            {
                System.out.println(sRs.getString(i));
            }
        }
    }
}while(stmt.getMoreResults());
sCallStmt.close();
```

3.6 JDBC and Failover

This section explains how to use the Failover feature in an Altibase JDBC application.

3.6.1 What is a Failover?

When a failure occurs on the database server and the connection is disconnected, a Failover is a feature which enables the application to immediately establish a connection to another server to continue the execution of the previously executed operation.

Failover can operate in the following two ways:

- CTF(Connection Time Failover)

CTF attempts to connect to another server when an attempt to connect to the database is unsuccessful. CTF can occur when the connect method of a Connection object is called.

- STF(Session Time Failover)

STF connects to another server and continuously executes the user-specified operation when a connection error occurs before the result of a SQL statement is received from the server. STF can occur on the execution of all methods communicating with the server, excluding the connect method.

For further information on Failover, please refer to the "Failover" chapter of *Replication Manual*.

3.6.2 How to Use Failover

This section explains how to use the CTF and STF features in JDBC applications.

3.6.2.1 CTF

The CTF feature can be used by adding the following properties to the Properties object.

```
Properties sProps = new Properties();
sProps.put("alternateservers", "database1:20300, database2:20300");
sProps.put("connectionretrycount", "5");
sProps.put("connectionretrydelay", "2");
sProps.put("sessionfailover", "off");
sProps.put("failover_source", "DSN1");
```

For further information on each of the properties, please refer to "Connection Information" of Chapter 1.

3.6.2.2 STF

The STF feature can be used by additionally setting "SessionFailover=on" to the properties which set the CTF feature.

In communication situations other than attempting to establish connection to the database server, the client first processes CTF and restores the connection when it detects server failure. Thereafter, the client executes the callback function registered by the user and raises a Failover Success Exception for the user to acknowledge that a Failover has occurred. If Failover fails to every server, the driver throws the Exception which originally occurred.

The following is an interface for the Failover callback function written by the user.

```
public interface AltibaseFailoverCallback
{
    public final static class Event
    {
        public static final int BEGIN     = 0;
        public static final int COMPLETED = 1;
        public static final int ABORT     = 2;
    }
    public final static class Result
    {
        public static final int GO   = 3;
        public static final int QUIT = 4;
    }
    int failoverCallback(Connection aConnection,
                         Object     aAppContext,
                         int        aFailoverEvent);
};
```

The following is a code example which shows the process of a user registering and freeing a Failover callback function.

```
public class  UserDefinedFailoverCallback implements AltibaseFailoverCallback
{
    ...

    public int failoverCallback(Connection aConnection,
                                Object     aAppContext,
                                int        aFailoverEvent)
    {
        // User Defined Code
        // Must return either Result.GO or Result.QUIT.
    }

    ...
}
```

If the Failover callback function written by the user is called by the JDBC driver, one of the Event constants included in the above AltibaseFailoverCallback interface is passed to aFailoverEvent, which is the third argument of the callback function. The meaning of each Event constant is as follows:

- Event.BEGIN: Session Failover is started

- Event.COMPLETED: Session Failover has succeeded
- Event.ABORT: Session Failover has failed

The Result constants included in the AltibaseFailoverCallback interface are values which can be returned by the callback function written by the user. If values other than Result constants are returned from the callback function, Failover does not operate normally.

- Result.GO: If this constant value is returned from the callback function, the JDBC driver continually runs the next process of STF.
- Result.QUIT: If this constant value is returned from the callback function, the JDBC driver terminates the STF process.

The following is a code example of an object which can be used as the second argument of the Failover callback function written by the user.

```
public class UserDefinedAppContext
{
    // User Defined Code
}
```

If there is a need to use information of an application implemented by the user during the STF process, the object to be passed to the callback function while registering the Failover callback function can be specified. If this object is specified as the second argument of the registerFailoverCallback method which registers the callback function, this object is passed when the callback function is actually called. The following is an example which depicts this process in code.

```
// Create a user-defined callback function object.
UserDefinedFailoverCallback sCallback = new UserDefinedFailoverCallback();
// Create a user-defined application information object
UserDefinedAppContext sAppContext = new UserDefinedAppContext();

...

Connection sCon = DriverManager.getConnection(sURL, sProp);

// Register the callback function with the user-defined application object
((AltibaseConnection)sCon).registerFailoverCallback(sCallback, sAppContext);

...

// Free the callback function
((AltibaseConnection)sCon).deregisterFailoverCallback();
```

3.6.3 Code Examples

This is a code example which implements a callback function for STF.

The following is an example of a simple code which is regardless of various circumstances; therefore, it should be noted that it cannot be used as it is in user applications.

```
public class MyFailoverCallback implements AltibaseFailoverCallback
{
    public int failoverCallback(Connection aConnection, Object aAppContext,int aFailoverEvent)
    {
        Statement sStmt = null;
        ResultSet sRes = null;

        switch (aFailoverEvent)
        {
            // Necessary operations before starting Failover on the user application logic can be executed.
            case Event.BEGIN:
                System.out.println("Failover Started .... ");
                break;
            // Necessary operations after completing Failover on the user application logic can be executed.
            case Event.COMPLETED:
                try
                {
                    sStmt = aConnection.createStatement();
                }
                catch( SQLException ex1 )
                {
                    try
                    {
                        sStmt.close();
                    }
                    catch( SQLException ex3 )
                    {
                    }
                    return Result.QUIT;
                }

                try
                {
                    sRes = sStmt.executeQuery("select 1 from dual");
                    while(sRes.next())
                    {
                        if(sRes.getInt(1) == 1 )
                        {
                            break;
                        }
                    }
                }
                catch ( SQLException ex2 )
                {
                    try
                    {
                        sStmt.close();
                    }
                    catch( SQLException ex3 )
                    {
                    }
                    // Terminates the Failover process.
                    return Result.QUIT;
                }
                break;
        }
        // Continues the Failover process.
        return Result.GO;
    }
}
```

The following is a code example which checks whether or not STF was successful. Whether STF succeeded or failed can be confirmed by checking whether ErrorCode of SQLException is identical to Validation.FAILOVER_SUCCESS. The Failover validation code is inserted inside the

while loop because the operation which was previously under execution must be executed again, even if Failover succeeds.

```
// Must be implemented so that the operation to be executed can be re-executed..
// The while loop has been used in this case.
while (true)
{
    try
    {
        sStmt = sConn.createStatement();
        sRes = sStmt.executeQuery("SELECT C1 FROM T1");
        while (sRes.next())
        {
            System.out.println("VALUE : " + sRes.getString(1));
        }
    }
    catch (SQLException e)
    {
        // Whether or not the Failover has succeeded.
        if (e.getErrorCode() == AltibaseFailoverCallback.FailoverValidation.FAILOVER_SUCCESS)
        {
            // Since Failover has succeeded, Exception is ignored and the process is continued.
            continue;
        }
        System.out.println("EXCEPTION : " + e.getMessage());
    }
    break;
}
```

3.7 JDBC Escapes

The JDBC specification provides the escape syntax for the JDBC application to understand vendor specific SQL for database products. The JDBC driver converts a SQL statement which includes the escape syntax to a native SQL statement for its database.

The following table is an organization of SQL statements which include the escape syntax supported by the JDBC specification and SQL statements converted by the JDBC driver for use in Altibase.

Type	SQL statements supported in the JDBC specification	SQL statements converted for use in Altibase
ESCAPE	SELECT cVARCHAR FROM t1 WHERE cVARCHAR LIKE '%a|%b%' {escape '|'}	SELECT cVARCHAR FROM t1 WHERE cVARCHAR LIKE '%a|%b%' escape '|'
FN	SELECT {fn concat('concat', 'test')} FROM dual	SELECT concat('concat', 'test') FROM dual
DTS	UPDATE t1 SET cDATE = {d '1234-12-30'}	UPDATE t1 SET cDATE = to_date('1234-12-30', 'yyyy-MM-dd')
	UPDATE t1 SET cDATE = {t '12:34:56'}	UPDATE t1 SET cDATE = to_date('12:34:56', 'hh24:mi:ss')
	UPDATE t1 SET cDATE = {ts '2010-01-23 12:23:45'}	UPDATE t1 SET cDATE = to_date('2010-01-23 12:23:45', 'yyyy-MM-dd hh24:mi:ss')
	UPDATE t1 SET cDATE = {ts '2010-11-29 23:01:23.971589'}	UPDATE t1 SET cDATE = to_date('2010-11-29 23:01:23.971589', 'yyyy-MM-dd hh24:mi:ss.ff6')
CALL	{call p1()}	execute p1()
	{? = call p2(?)}	execute ? := p2(?)
OJ	SELECT * FROM {oj t1 LEFT OUTER JOIN t2 ON t1.cINT = t2.cINT}	SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.cINT = t2.cINT

3.8 How to Use ResultSet

This section explains the ResultSet types supported by the Altibase JDBC driver and offers instructions on how to use it.

3.8.1 Creating ResultSet

A ResultSet is created when a query statement is executed on the database, and it corresponds to the ResultSet object of JDBC.

The following methods create the ResultSet object in JDBC.

```
public Statement createStatement(int aResultSetType, int aResultSetConcurrency) throws SQLException;

public Statement createStatement(int aResultSetType, int aResultSetConcurrency, int aResultSetHoldability)
throws SQLException;

public PreparedStatement prepareStatement(String aSql, int aResultSetType, int aResultSetConcurrency) throws
SQLException;

public PreparedStatement prepareStatement(String aSql, int aResultSetType, int aResultSetConcurrency, int
aResultSetHoldability) throws SQLException;

public CallableStatement prepareCall(String aSql, int aResultSetType, int aResultSetConcurrency) throws
SQLException

public CallableStatement prepareCall(String aSql, int aResultSetType, int aResultSetConcurrency, int
aResultSetHoldability) throws SQLException;
```

3.8.2 ResultSet Types

The ResultSet object of JDBC manages and retains the cursor which points to the current position within the result set. The cursor of a basic ResultSet object is not updateable and only moves forward; however, a scrollable and updateable ResultSet object can be created with the use of options.

The following are ResultSet object types available for user specification.

- TYPE_FORWARD_ONLY

Unscrollable; the cursor can be moved only forward. Data of the ResultSet is determined at the point in time at which the cursor opens in the database server.

- TYPE_SCROLL_INSENSITIVE

Scrollable; the cursor can be moved forward, backwards, or moved to a specified location. Data of the ResultSet is determined at the point in time at which the cursor opens in the database server. Memory can become scarce, due to caching the ResultSet retrieved from the

server on the client.

- TYPE_SCROLL_SENSITIVE

Scrollable; the cursor can be moved forward, backwards, or moved to a specified location. The ResultSet is determined at the point in time at which the cursor opens in the database server; however, data within the ResultSet is determined at the point in time at which the client retrieves or updates it.

3.8.3 Concurrency

This option determines whether or not to allow updates through the ResultSet object. One of the following two constants is available for use:

- CONCUR_READ_ONLY

Does not allow updates; the default value.

- CONCUR_UPDATABLE

Allows updates with the ResultSet object.

3.8.4 Holdability

This option determines whether or not to retain the ResultSet object after the transaction has been committed. One of the following two constants are available for use:

- CLOSE_CURSORS_AT_COMMIT

The cursor is closed when the transaction is committed.

- HOLD_CURSORS_OVER_COMMIT

The cursor is left open, even if the transaction is committed. If the transaction has been committed at least once after the cursor has been opened, the cursor is left open during future commit and rollback operations. If the transaction has not been committed even once since the cursor has been opened, however, the cursor is closed when the transaction is rolled back.

3.8.4.1 Notes

- Since the JDBC driver caches as many number of rows as the value set for FetchSize for the ResultSet object on the client, data left in the cache can be retrieved by the application, even if the cursor is closed. If you want the application to immediately detect that the cursor has been closed, set FetchSize to 1.

- The default value of Holdability for the Altibase JDBC driver is CLOSE_CURSORS_AT_COMMIT, and is different from the default value for the JDBC specification, HOLD_CURSORS_OVER_COMMIT.

Open ResultSet objects must be closed prior to switching the autocommit mode with the setAutoCommit() method in a session where Holdability is HOLD_CURSORS_OVER_COMMIT. The following is a code example which raises an error.

```
sCon = getConnection();
sStmt = sCon.createStatement();
byte[] br;
byte[] bb = new byte[48];
for(byte i = 0; i < bb.length;i++) bb[i] = i;

sCon.setAutoCommit(false);

sStmt.executeUpdate("insert into Varbinary_Tab values(null)");
sCon.commit();

sPreStmt = sCon.prepareStatement("update Varbinary_Tab set VARBINARY_VAL=?");
sPreStmt.setObject(1, bb, java.sql.Types.VARBINARY);
sPreStmt.executeUpdate();

sRS = sStmt.executeQuery("Select VARBINARY_VAL from Varbinary_Tab");
sRS.next();
br = sRS.getBytes(1);

sCon.commit();
sCon.setAutoCommit(true); -> 1
```

The following exception is raised at 1.

```
java.sql.SQLException: Several statements still open
    at Altibase.jdbc.driver.ex.Error.processServerError(Error.java:320)
    at Altibase.jdbc.driver.AltibaseConnection.setAutoCommit(AltibaseConnection.java:988)
    at HodabilityTest.testHoldability(HodabilityTest.java:46)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:616)
```

sRs.close() must be called before sCon.setAutoCommit(true) to not raise an exception.

- The client session must be in Non-Autocommit mode or the clientside_auto_commit connection attribute must be set to on to use a ResultSet object whose Holdability type is HOLD_CURSORS_OVER_COMMIT. If the clientside_auto_commit connection attribute is set to on, the Holdability type is automatically changed to HOLD_CURSORS_OVER_COMMIT.

3.8.4.2 Examples

```
Statement sUpdStmt = sConn.prepareStatement("UPDATE t1 SET val = ? WHERE id = ?");
Statement sSelStmt = sConn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY, ResultSet.HOLD_CURSORS_OVER_COMMIT);
ResultSet sRS = sSelStmt.executeQuery("SELECT * FROM t1");
while (sRS.next())
{
```

```
        // TODO : set parameters

        sUpdStmt.execute();
        sConn.commit();
    }
    sRS.close();
```

3.8.5 Restrictions

To use an Updatable ResultSet or a Scrollable ResultSet, a SELECT query statement which retrieves the ResultSet is restricted in the following ways:

To use an Updatable ResultSet,

- Only one table can be specified in the FROM clause.
- Only pure columns can be specified in the SELECT list; expressions or functions cannot be included. Columns with a NOT NULL constraint, and without a default value must be included in the SELECT list.

To use a Scrollable-Sensitive ResultSet,

- Only one table can be specified in the FROM clause.

When executing PSM, only ResultSet objects of the default type are available for use. If the user specifies an option which is not of the default type, the option is ignored.

Since for a ResultSet object which is CONCUR_UPDATABLE and TYPE_SCROLL_SENSITIVE, one more Statement is used within the JDBC driver, it can easily exceed the limited number of Statements; therefore, the maximum number of Statements must be set for occasions on which such ResultSet types are used a lot.

Since an updateable and scrollable ResultSet contains a large amount of data, its memory usage is higher than a forward only ResultSet. A large ResultSet can cause memory to become scarce, so its use is not recommended.

The characteristics of the ResultSet are determined by the ResultSet type, concurrency type and holdability type described above. The user can specify random combinations for these three values; however, depending on the query statement that generates the ResultSet, the user-defined combination can be invalid. In this case, the driver does not raise an exception, but converts it to a valid combination. In the following example, the invalid types on the left side are automatically converted to the valid types on the right side.

- TYPE_SCROLL_SENSITIVE -> TYPE_SCROLL_INSENSITIVE
- CONCUR_UPDATABLE -> CONCUR_READ_ONLY
- HOLD_CURSORS_OVER_COMMIT -> CLOSE_CURSORS_AT_COMMIT

When a conversion is made internally, whether or not a conversion has occurred can be confirmed through warnings.

3.8.6 Detecting Holes

A ResultSet object of the TYPE_SCROLL_SENSITIVE type retrieves the newest data from the server when performing a FETCH. Therefore, a row which was visible when the cursor opened can become invisible as the row is scrolled. For example, if a row in a ResultSet object is deleted by another Statement, the row is no longer visible in the ResultSet object. Such an invisible row is called a Hole.

The following is a code example which detects Holes in JDBC.

```
while (sRS.next())
{
    if (sRS.rowDeleted())
    {
        // HOLE DETECTED!!!
    }
    else
    {
        // do something ...
    }
}
```

Valid data cannot be obtained from a Hole, and a ResultSet returns one of the following values for a Hole: a SQL data type NULL, a reference type NULL, or the value 0.

3.8.7 Fetch Size

When retrieving data for the ResultSet object from the server, the Altibase JDBC driver retrieves multiple rows at once, instead of retrieving one row each time, and caches them in the client to enhance performance. This is called a prefetch, and the number of rows to be fetched can be set with the setFetchSize() method of the Statement object.

```
public void setFetchSize(int aRows) throws SQLException;
```

A value between the range of 0 to 2147483647 can be set for the Altibase JDBC driver. The JDBC specification defines that an exception must be raised when a value outside of this range is specified; however, the Altibase JDBC driver does not raise an exception and ignores it, for the sake of convenience.

If the value is set to 0, the Altibase server voluntarily determines the size to return to the client in one go. In this case, the number of rows to be returned differ, according to the size of a row.

The FetchSize value is especially important for the Scroll-Sensitive ResultSet. When the user

retrieves data from a Scroll-Sensitive ResultSet, the driver returns the prefetched rows first. Even if data of the database has been updated, as long as the row exists in the prefetched cache, data of the cache is returned to the user. If the user wants to see the newest data of the database, FetchSize should be set to 1. By doing so, however, the frequency of retrieving data from the server increases and performance can be lowered.

3.8.8 Refreshing Rows

With the refreshRow() method of the ResultSet object, data which has been previously retrieved from the server can be re-fetched, without executing the SELECT statement. The refreshRow() method retrieves as many number of rows as the value set for FetchSize, based on the current row. To use this method, a cursor must be pointing to any row in the ResultSet.

This method operates when the ResultSet object is of the following types:

- TYPE_SCROLL_SENSITIVE & CONCUR_UPDATABLE
- TYPE_SCROLL_SENSITIVE & CONCUR_READ_ONLY

If this method is called for a TYPE_FORWARD_ONLY type, an exception is raised; for a TYPE_SCROLL_INSENSITIVE type, nothing happens.

3.9 Atomic Batch

The Altibase JDBC driver not only guarantees the atomicity of batch operations, but also supports fast INSERT operations of bulk data through the Atomic Batch feature.

This section explains how to use the Atomic Batch feature which the Altibase JDBC driver supports.

3.9.1 How to Use Atomic Batch

In order to use the Atomic Batch feature, you must first create the PreparedStatement object and then cast the object to the AltibasePreparedStatement class type in java programming.

The following method, setAtmoicBatch(), can be used to enable the Atomic Batch feature.

```
public void setAtomicBatch(boolean aValue) throws SQLException
```

To confirm whether or not Atomic Batch is set for the PreparedStatement object, call the getAtomicBatch() method as below.

```
public boolean getAtomicBatch()
```

3.9.2 Restrictions

When using the Atomic Batch feature in Altibase, the following restrictions apply:

- Only supports simple INSERT statements. Consistency for complex INSERT statements or DML statements, such as UPDATE, DELETE, etc., cannot be assured.
- If a trigger fires with Each Statement as the unit, the trigger fires only once.
- SYSDATE operates only once.

3.9.3 Examples

```
......
Connection con = sConn = DriverManager.getConnection(aConnectionStr, mProps);
Statement stmt = con.createStatement();
try
{
stmt.execute("Drop table " + TABLE_NAME); } catch (SQLException e) { }
stmt.execute("create table " + TABLE_NAME + "(c1 VARCHAR (1000))");

PreparedStatement sPrepareStmt = con.prepareStatement("insert into " +
TABLE_NAME + " values(?)");
((AltibasePreparedStatement)sPrepareStmt).setAtomicBatch(true);

for(int i = 1; i <= MAX_RECORD_CNT; i++)
```

```
{
sPrepareStmt.setString(1, String.valueOf(i % 50));
sPrepareStmt.addBatch();

if(i%BATCH_SIZE == 0)
{
sPrepareStmt.executeBatch();
con.commit();
}
}
con.commit();
}
catch (SQLException e)
{
System.out.println(e.getMessage());
}
.....
```

3.10 Date, Time, Timestamp

This section explains the meanings of Date, Time, and Timestamp which are DATE types, and describes the data conversion range supported by the Altibase JDBC driver.

3.10.1 Meanings

- Date: Expresses only the date
- Time: Expresses the time(the date can be included)
- Timestamp: Expresses the date, time, seconds and further subdivisions of time

3.10.2 Conversion Table

The following table shows the formats that are processed by the Altibase JDBC driver according to the object type passed to the setObject method.

	String	Date	Time	Timestamp
setObject (DATE)	2134-12-23 00:00:00.0 An error is raised if the user inputs values to the hour:minute:second. The driver sets it to 0.	2134-12-23 00:00:00.0 The values input to the hour:minute:second are ignored by the driver.	SQLException: UNSUPPORTED_TYPE_CONVERSION	2134-12-23 12:34:56.123456
setObject (TIME)	1970-01-01 12:34:56.0 An error is raised if the user inputs values to the year:month:date or a value to the nanosecond. The driver sets it to the standard year:month:date.	2134-12-23 12:34:56.0	2134-12-23 12:34:56.0	2134-12-23 12:34:56.0
setObject (TIMESTAMP)	2134-12-23 12:34:56.123456	2134-12-23 00:00:00.0 The values input to the hour:minute:second are ignored by the driver.	SQLException: UNSUPPORTED_TYPE_CONVERSION	2134-12-23 12:34:56.123456

	String	Date	Time	Timestamp
setString()	Must be input in the format set for the DATE_FORMAT property.	-	-	-
setDate()	-	2134-12-23 00:00:00.0 The values input to the hour:minute:second are ignored by the driver.	-	-
setTime()	-	-	2134-12-23 12:34:56.0	-
setTimestamp()	-	-	-	2134-12-23 12:34:56.123456

The following table shows the values returned from the DATE type value (1234-01-23 12:23:34.567123) stored in the database with the getDate(), getTime(), and getTimestamp() methods.

Function	Return Value
getDate()	1234-01-23 00:00:00.0
getTime()	1234-01-23 12:23:34.0
getTimestamp()	1234-01-23 12:23:34.567123

3.11 GEOMETRY

This section offers instructions on how to use GEOMETRY type data provided by Altibase in JDBC applications.

3.11.1 How To Use GEOMETRY

GEOMETRY type data can be used with the byte array in Altibase JDBC applications.

When inserting data(including NULL) to a GEOMETRY type column in the database with the IN parameter of PreparedStatement, the data type must be specified with the AltibaseTypes.GEOMETRY constant.

For further information on how to directly write GEOMETRY type data in a query statement, please refer to *Spatial SQL Reference*.

3.11.2 Examples

The following is code example which inserts data to a GEOMETRY type column in a JDBC application.

```
int sSize = ... ;
byte[] sGeometryData = new byte[sSize];

Connection sConn = ... ;
PreparedStatement sPstmt = sConn.prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");
sPstmt.setObject(1, sGeometryData, AltibaseTypes.GEOMETRY);
sPstmt.executeQuery();

...
```

3.12 LOB

This sections offers instructions on how to use LOB type data provided by Altibase in a JDBC application.

3.12.1 Prerequisites

- Altibase supports the LOB data types, BLOB and CLOB, and each can have the maximum size of 2Gbytes.

To manipulate LOB data, the autocommit mode of a session must satisfy one of the following conditions.

- The autocommit mode of a session must be disabled with setAutoCommit(false) of the Connection object and the user must manually control transactions.
- Clientside_auto_commit must be set to on to enable the JDBC driver to control the autocommit operations of transactions.

3.12.2 Using BLOB

How to manipulate BLOB data in a JDBC application is shown in the following code examples.

3.12.2.1 Writing BLOB Data Through the PreparedStatement Object

The following statement creates the table used in the examples.

```
CREATE TABLE TEST_TABLE ( C1 BLOB );
```

Using the setBinaryStream method with an InputStream object

```
InputStream sInputStream = ...
long sLength = ...

...

PreparedStatement sPstmt = connection().prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");

...

sPstmt.setBinaryStream(1, sInputStream, sLength);

...

sPstmt.execute();

...
```

Using the setBinaryStream method with an OutputStream object

```
byte[] sBuf = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT * FROM TEST_TABLE FOR UPDATE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    Blob sBlob = sPstmt.getBlob(1);
    OutputStream sOutputStream = sBlob.setBinaryStream(1);
    sOutputStream.write(sBuf);
    sOutputStream.close();
    ...
}

...

sPstmt.execute();

...
```

Using the setBlob method with a Blob object

```
java.sql.Blob sBlob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");

...

sPstmt.setBlob(1, sBlob);

...

sPstmt.execute();

...
```

Using the setObject method with a Blob object

```
java.sql.Blob sBlob = ...
...

PreparedStatement sPstmt = connection().prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");

...

sPstmt.setObject(1, sBlob);

...

sPstmt.execute();

...
```

Specifying a SQL type for the setObject method

```
java.sql.Blob sBlob = ...
...
```

```
PreparedStatement sPstmt = connection().prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");

...

sPstmt.setObject(1, sBlob);

...

sPstmt.execute();

...
```

3.12.2.2 Writing BLOB Data Through the ResultSet object

The following statement creates the table used in the examples.

```
CREATE TABLE BLOB_TABLE ( BLOB_COLUMN BLOB );
```

Using the updateBinaryStream method with an InputStream object

```
InputStream sInputStream = ...
long sLength = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT BLOB_COLUMN FROM
BLOB_TABLE", ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    sRs.updateBinaryStream(1, sInputStream, sLength);
    sRs.updateRow();
    ...
}

...
```

Using the updateBlob method with a Blob object

```
java.sql.Blob sBlob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT BLOB_COLUMN FROM
BLOB_TABLE", ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    sRs.updateBlob(1, sBlob), ;
    sRs.updateRow();
    ...
}

...
```

Using the updateObject method with a Blob object

```
java.sql.Blob sBlob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT BLOB_COLUMN FROM
BLOB_TABLE", ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    sRs.updateObject(1, sBlob);
    sRs.updateRow();
    ...
}

...
```

Specifying the SQL type for the updateObject method

```
java.sql.Blob sBlob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT BLOB_COLUMN FROM
BLOB_TABLE", ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    sRs.updateObject(1, sBlob, AltibaseTypes.BLOB);
    sRs.updateRow();
    ...
}

...
```

3.12.2.3 Updating BLOB Data with the SELECT … FOR UPDATE Statement

```
byte[] sBytes = new byte[sLength];
...

PreparedStatement sPstmt = connection().prepareStatement("SELECT BLOB_COLUMN FROM
BLOB_TABLE FOR UPDATE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    Blob sBlob = sRs.getBlob(1);
    sBlob.setBytes(0, sBytes);
    ...
}

...
```

3.12.2.4 Reading BLOB Data

Using the getBinaryStream method with an InputStream object

```
...

PreparedStatement sPstmt = connection().prepareStatement("SELECT BLOB_COLUMN FROM
BLOB_TABLE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    InputStream sInputStream = sRs.getBinaryStream(1);
    ...
}

...
```

Using the getBlob method with an InputStream object

```
...

PreparedStatement sPstmt = connection().prepareStatement("SELECT BLOB_COLUMN FROM
BLOB_TABLE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    Blob sBlob = sRs.getBlob(1);
    InputStream sInputStream = sBlob.getBinaryStream();
    ...
}

...
```

Using the getBlob method with a byte array

```
...
final int sReadLength = 100;

PreparedStatement sPstmt = connection().prepareStatement("SELECT BLOB_COLUMN FROM
BLOB_TABLE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    Blob sBlob = sRs.getBlob(1);
    long sRemains = sBlob.length();
    long sOffset = 0;
    while(sRemains > 0)
    {
        byte[] sReadBytes = sBlob.getBytes(sOffset, sReadLength);
        sRemains -= sReadBytes.length;
        sOffset += sReadBytes.length;
        ...
    }
    ...
}
```

```
...
```

3.12.2.5 Altering BLOB Data

Truncation

```
Statement sStmt = ...

ResultSet sRs = sStmt.executeQuery("SELECT * FROM t1 FOR UPDATE");

while(sRs.next())
{
    ...
    int sLength = ... ;
    Blob sBlob = sRs.getBlob(2);

    // After executing this method
    // sBlob.length() == sLength
    sBlob.truncate(sLength);
}

...
```

3.12.3 Using CLOB Data

How to use CLOB data in a JDBC application is shown in the following code examples.

3.12.3.1 Writing CLOB data Through PreparedStatement

The following statement creates the table used in the examples.

```
CREATE TABLE TEST_TABLE ( C1 CLOB );
```

Using the setCharacterStream method with a Reader object

```
Reader sReader = ...
long sLength = ...

...


PreparedStatement sPstmt = connection().prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");

...


sPstmt.setCharacterStream(1, sReader, sLength);


...


sPstmt.execute();


...
```

Using the setCharacterStream method with a Writer object

```
char[] sBuf = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT * FROM TEST_TABLE FOR UPDATE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    Clob sClob = sPstmt.getClob(1);
    Writer sWriter = sClob.setCharacterStream(1);
    sWriter.write(sBuf);
    sWriter.close();
    ...
}

...

sPstmt.execute();

...
```

Using the setClob method with a Clob object

```
java.sql.Clob sClob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");

...


sPstmt.setClob(1, sClob);


...


sPstmt.execute();


...
```

Using the setObject method with a Clob object

```
java.sql.Clob sClob = ...
...

PreparedStatement sPstmt = connection().prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");

...


sPstmt.setObject(1, sClob);


...


sPstmt.execute();
```

```
...
```

Specifying a SQL type for the setObject method

```
java.sql.Clob sClob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("INSERT INTO TEST_TABLE VALUES (?)");

...


sPstmt.setObject(1, sClob, AltibaseTypes.Clob);


...


sPstmt.execute();


...
```

3.12.3.2 Writing CLOB data Through the ResultSet Object

The following statement creates the table used in the examples.

```
CREATE TABLE CLOB_TABLE ( CLOB_COLUMN CLOB );
```

Using the updateCharacterStream method with a Reader object

```
Reader sReader = ...
long sLength = ... // The length of source from which Reader is linked

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT CLOB_COLUMN FROM
CLOB_TABLE", ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
      ...
      sRs.updateCharacterStream(1, sReader, sLength);
      sRs.updateRow();
      ...
}

...
```

Using the updateClob method with a Clob object

```
java.sql.Clob sClob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT CLOB_COLUMN FROM
CLOB_TABLE", ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
```

```
ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    sRs.updateClob(1, sClob);
    sRs.updateRow();
    ...
}

...
```

Using the updateObject method witha Clob object

```
java.sql.Clob sClob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT CLOB_COLUMN FROM
CLOB_TABLE", ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    sRs.updateObject(1, sClob);
    sRs.updateRow();
    ...
}

...
```

Specifying a SQL type for the updateObject method

```
java.sql.Clob sClob = ...

...

PreparedStatement sPstmt = connection().prepareStatement("SELECT CLOB_COLUMN FROM
CLOB_TABLE", ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    sRs.updateObject(1, sClob, AltibaseTypes.CLOB);
    sRs.updateRow();
    ...
}

...
```

3.12.3.3 Inserting CLOB data With the SELECT … FOR UPDATE Statement

```
...

String sStr = ... ;
PreparedStatement sPstmt = connection().prepareStatement("SELECT CLOB_COLUMN FROM
CLOB_TABLE FOR UPDATE");

ResultSet sRs = sPstmt.executeQuery();
```

```
while(sRs.next())
{
    ...
    Clob sClob = sRs.getClob(1);
    sClob.setString(0, sStr);
    ...
}

...
```

3.12.3.4 Reading CLOB Data

Using the getCharacterStream method with a Reader object

```
...

PreparedStatement sPstmt = connection().prepareStatement("SELECT CLOB_COLUMN FROM
CLOB_TABLE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    Reader sReader = sRs.getCharacterStream(1);
    ...
}

...
```

Using the getClob method with a Reader object

```
...

PreparedStatement sPstmt = connection().prepareStatement("SELECT CLOB_COLUMN FROM
CLOB_TABLE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
    Clob sClob = sRs.getClob(1);
    Reader sReader = sClob.getCharacterStream();
    ...
}

...
```

Using the getClob method with a String object

```
...
final int sReadLength = 100;

PreparedStatement sPstmt = connection().prepareStatement("SELECT CLOB_COLUMN FROM
CLOB_TABLE");

ResultSet sRs = sPstmt.executeQuery();

while(sRs.next())
{
    ...
```

```
        Clob sClob = sRs.getClob(1);
        long sRemains = sClob.length();
        long sOffset = 0;
        while(sRemains > 0)
        {
            String sStr = sClob.getSubString(sOffset, sReadLength);
            sRemains -= sStr.length;
            sOffset += sStr.length;
            ...
        }
        ...
}

...
```

3.12.3.5 Altering CLOB Data

Truncation

```
Statement sStmt = ...

ResultSet sRs = sStmt.executeQuery("SELECT * FROM t1 FOR UPDATE");

while(sRs.next())
{
    ...
    int sLength = ... ;
    Clob sClob = sRs.getClob(2);

    // After executing this method
    // sClob.length() == sLength
    sClob.truncate(sLength);
}

...
```

3.12.4 Freeing Resources

For JDBC applications which obtain data through a large number of LOB objects, the obtained LOB objects must be freed. The LOB object must be freed specifically, regardless of whether or not the transaction is committed.

The following is a code example which frees a Blob object.

```
...

Blob sBlob = sRs.getBlob(1);

// Freeing Lob Locator
((Altibase.jdbc.driver.AltibaseLob)sBlob).free();

...
```

Further operations cannot be executed on an object if a Blob object is freed with the free method, since the corresponding Lob Locator is freed from the server.

The following is a code example which frees a Clob object.

```
...

Clob sClob = sRs.getClob(1);

// Freeing Lob Locator
((Altibase.jdbc.driver.AltibaseLob)sClob).free();

...
```

As for Blob objects, further operations cannot be executed on an object if a Clob object is freed with the free method, since the corresponding Lob Locator is freed from the server.

The following is a code example which frees the BlobInputStream and BlobOutputStream objects.

```
InputStream sInputStream = sRs.getBinaryStream(1);

// Freeing Lob Locator
((Altibase.jdbc.driver.BlobInputStream)sInputStream).freeLocator();


CallableStatement sCallStmt = aConn.prepareCall("INSERT INTO TEST_TABLE VALUES (?)");
sCallStmt.registerOutParameter(1, Types.BLOB);
sCallStmt.execute();

Blob sBlob = sCallStmt.getBlob(1);
OutputStream sOutputStream = sBlob.setBinaryStream(1);

// Freeing Lob Locator
((Altibase.jdbc.driver.BlobOutputStream)sOutputStream).freeLocator();
```

Further operations cannot be executed on an object if the BlobInputStream object or the BlobOutputStream object is freed with the freeLocator method, since the corresponding Lob Locator is freed from the server.

The following is a code example which frees the ClobReader and ClobWriter objects.

```
Reader sClobReader = sRs.getCharacterStream(1);

// Freeing Lob Locator
((Altibase.jdbc.driver.ClobReader)sClobReader).freeLocator();


CallableStatement sCallStmt = aConn.prepareCall("INSERT INTO TEST_TABLE VALUES (?)");
sCallStmt.registerOutParameter(1, Types.CLOB);
sCallStmt.execute();

Clob sClob = sCallStmt.getClob(1);
Writer sClobWriter = sClob.setCharacterStream(1);

// Freeing Lob Locator
((Altibase.jdbc.driver.ClobWriter)sClobWriter).freeLocator();
```

Further operations cannot be executed on an object if the ClobReader object or the ClobWriter object is freed with the freeLocator method, since the corresponding Lob Locator is freed from the server.

3.12.5 Restrictions

Even if clientside_auto_commit is set to on to enable the JDBC driver to control the autocommit operations of transactions the following restrictions still apply to the manipulation of LOB data.

If LOB data retrieved from the ResultSet object(cursor) is used with the executeUpdate() method of another Statement before the cursor is closed, no more fetch operations are possible from the cursor since the Lob locator is freed. The following is a code example which raises such an error.

```
PreparedStatement sPreStmt =
      sCon.prepareStatement( "INSERT INTO TEST_TEXT " +
                                    "VALUES ( ?, ?, ?, ? )" );
Statement sStmt = sCon.createStatement();
ResultSet sRS = sStmt.executeQuery( "SELECT ID, TEXT " +
                                                "    FROM TEST_SAMPLE_TEXT " );
while ( sRS.next() ) -> 2
{
      sID     = sRS.getInt( 1 );
      sClob = sRS.getClob( 2 );
      switch ( sID )
      {
            case 1 :
                  sPreStmt.setInt(       1, 1 );
                  sPreStmt.setString( 2, "Altibase Greetings" );
                  sPreStmt.setClob(      3, sClob );
                  sPreStmt.setInt(       4, (int)sClob.length() );
                  break;
            case 2 :
                  sPreStmt.setInt(       1, 2 );
                  sPreStmt.setString( 2, "Main Memory DBMS" );
                  sPreStmt.setClob(      3, sClob );
                  sPreStmt.setInt(       4, (int)sClob.length() );
                  break;
            default :
                  break;
      }
      sPreStmt.executeUpdate(); -> 1
}
```

1: If sPreStmt.executeUpdate() is called while ResultSet sRS is open, the JDBC driver automatically commits transactions and by doing so, the Lob locator of sClob is freed.

2: An exception can be raised at sRs.next() since the Lob locator is freed.

Thus, when manipulating LOB data in such a logic as above, the autocommit mode of a session must first be disabled by calling setAutoCommit(false).

3.13 Controlling Autocommit

The autocommit mode of a session can be set with the auto_commit connection attribute or the setAutoCommit method of the JDBC Connection object for Altibase JDBC applications. If autocommit is enabled with auto_commit=true or the setAutoCommit(true) method, the Altibase server automatically commits transactions.

Autocommit can also be enabled with the clientside_auto_commit connection attribute. If clientside_auto_commit is set to on, instead of the Altibase server, the JDBC driver automatically commits transactions.

When clientside_auto_commit is set to off, the autocommit mode of a session is determined by the setAutoCommit method.

To disable autocommit for a session, call setAutoCommit(false).

Calling setAutoCommit(false) on a client_auto_commit=on session disables autocommit and calling setAutoCommit(true) thereafter restores the session to the autocommit mode of the JDBC driver.

When autocommit is disabled, the user must manually commit or rollback with the commit() or rollback() method.

The following table sums up the above.

Autocommit Mode	How to Set
Server automatically commits transactions	auto_commit=true(or omitted) or setAutoCommit(true)
JDBC driver automatically commits transactions	auto_commit=true(or omitted) and Clientside_auto_commit=on
Autocommit is disabled	auto_commit=false or setAutoCommit(false)

3.14 JDBC Logging

JDBC Logging means to record all sorts of logs occurring in the Altibase JDBC driver, and the log can be recorded by using java.util.logging package. This section will discuss how to use and configure the JDBC logging.

3.14.1 Installing JDBC Logging

In order to record a log from the JDBC driver, the JDBC jar file with an added logging function should be used. Also, it should be used after activating a logging function in the ALTIBASE_JDBC_TRACE environment variable.

3.14.1.1 JRE Version

JRE1.5 or above should be installed in order to execute the JDBC logging and other library is not necessary.

3.14.1.2 Setting the CLASSPATH

To use the JDBC Logging, Altibase_t.jar file must be added to the CLASSPATH environment variable.

Ex) When using the bash shell in the Unix environment

```
export CLASSPATH=$ALTIBASE_HOME/lib/Altibase.jar:.:$CLASSPATH
```

3.14.1.3 Activating Logging

Global logging is activated without modifying a program by altering the ALTIBASE_JDBC_TRACE environment variables with JVM parameters. However, a client program should be re-started in order to apply the modified values of the ALTIBASE_JDBC_TRACE.

```
java -DALTIBASE_JDBC_TRACE=true ...
```

3.14.2 Instruction on JDBC Logging

3.14.2.1 Setting java.util.logging file

Setting java.util.logging can be executed in $JRE_HOME/lib/logging.properties or it can be separately configured on java.util.logging.config.file as follows.

```
java -Djava.util.logging.config.file=$ALTIBASE_HOME/sample/JDBC/Logging/logging.properties -
DALTIBASE_JDBC_TRACE=true ...
```

Altibase provide a logging.properties sample file in a directory of $ALTIBASE_HOME/sample/JDBC/Logging, and by using or referencing it the user can directly create a configuring file and use it Djava.util.logging.config.file property.

3.14.2.2 Logger type

A Logger is constructed with a tree structure and it is used to partially adjust setting or the amount of logs. The logger types supported by Altibase JDBC driver are as follows.

Logger type	Description
altibase.jdbc	Altibase JDBC messages(JDBC API call, such as connection, statement, prepared statement, etc.)
altibase.jdbc.pool	Messages regarding the connection pool
altibase.jdbc.rowset	ResultSet messages
altibase.jdbc.xa	xa messages
altibase.jdbc.failover	failover messages
altibase.jdbc.cm	CmChannel network packet messages

3.14.2.3 Logger Level

If a logger level is specified, the amount of logs can be concretely adjusted. The following table explicates the provided levels by Altibase JDBC driver, and the logs are left in detail more and more as it goes to FINEST from SEVER. If the CONFIG level is specified, logs of SEVERE, WARNING, INFO, CONFIG levels are left.

Logger level	Description
OFF	OFF is not record the log.
SEVERE	SQLException or when an inner error occurred, the relevant logs are recorded in the SEVERE level.
WARNING	SQLWarning is left the log in the WARNING level.
INFO	In the INFO level, the JDBC driver internally logs by monitoring specific objects.

Logger level	Description
CONFIG	This level is usually used to check what kind of a SQL statement is internally executed in the JDBC driver. In terms of the PreparedStatement, sql is displayed when preparing; however, in terms of a statement, the sql is displayed with milli sec unit when it is being executed.
FINE	This level leaves argument values and return values in the log when entering to the standard JDBC API. The amount of logs can be large since logs are left when entering to the API, and the time which is taken for connection or statement to close is additionally displayed.
FINEST	In the FINEST level, the packet information exchanged between JDBC driver and Altibase server is logged. It is the largest amount of logs.

3.14.2.4 logging.properties

The following is a logging.properties sample which leaves network packet logs with the log level of CONFIG. More information can be found in $ALTIBASE_HOME/sample/JDBC/Logging/logging.properties as well.

```
handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler --> FileHandler and ConsoleHandler added as basic handlers.

.level = CONFIG --> the root logger should be specified with the CONFIG.

# default file output is in same directory.
java.util.logging.FileHandler.level = CONFIG
java.util.logging.FileHandler.pattern = ./jdbc_trace.log
java.util.logging.FileHandler.limit = 10000000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.append = false
java.util.logging.FileHandler.formatter = Altibase.jdbc.driver.logging.SingleLineFormatter
  --> This is a section in which configuring the default setting of java.util.logging.FileHandler. The level of FileHandler is specified with CONFIG since only the sql information is needed.


java.util.logging.ConsoleHandler.level = CONFIG
java.util.logging.ConsoleHandler.formatter = Altibase.jdbc.driver.logging.SingleLineFormatter
  --> This is a section configuring java.util.logging.ConsoleHandler. SingleLineFormatter is used to print out logs in a line.

altibase.jdbc.level = CONFIG
altibase.jdbc.rowset.level = SEVERE
altibase.jdbc.cm.level = FINEST
altibase.jdbc.cm.handlers = Altibase.jdbc.driver.logging.MultipleFileHandler
#altibase.jdbc.cm.handlers = java.util.logging.FileHandler
  -->This is a section in which altibase jdbc logger is configured, and the rowset level is specified with SEVERE since the record set information needs to be excluded. The Network packet information should be loged;thus, the cm level is specified with FINEST. Moreover, in terms of the network packet, MultipleFileHandler was used as a handler in order to store a file per each session.

Altibase.jdbc.driver.logging.MultipleFileHandler.level = FINEST
Altibase.jdbc.driver.logging.MultipleFileHandler.pattern = ./jdbc_net_%s.log
Altibase.jdbc.driver.logging.MultipleFileHandler.limit = 10000000
Altibase.jdbc.driver.logging.MultipleFileHandler.count = 1
```

Altibase.jdbc.driver.logging.MultipleFileHandler.formatter = java.util.logging.XMLFormatter
--> By using jdbc_net_%s.log,it is configured that a file will be created per a session ID. Also, a log will be created in a XML formatter file by using XMLFormatter.

3.15 Hibernate

Altibase provides nonstandard SQL while Hibernate facilitates such provision of Altibase as supporting Dialect class. In order to interlock Altibase, the Altibase JDBC Driver should be configured and AltibaseDialect.class should be specified as well into configuration under Hibernate.

3.15.1 AltibaseDialect

Since the library that Hibernate officially provides does not include AltibaseDialect.class, AltibaseDialect.java file(include AltibaseLimitHandler.java as occasion arises) should be compiled and ported to a file Hibernate provides so that it can be available for use.

Refer to https://github.com/ALTIBASE/hibernate-orm/blob/master/ALTIBASE_DIALECT_PORTING.md for more details on how to use.

4 Tips & Recommendations

This chapter provides instructions for the efficient use of the Altibase JDBC driver.

4.1 Tips for Better Performance

The following tips should be kept in mind to enhance the performance of JDBC applications.

- It is recommended to use the Stream or Writer object when using LOB data in JDBC applications. If the size of the LOB data to be used is equal to or smaller than 8192 bytes, the Lob_Cache_Threshhold connection attribute must be set to an appropriate value.
- It is recommended to execute one operation on one Connection object. For example, if a multiple number of Statement objects are created in one Connection object and their operations are executed, this can induce performance loss.
- It is recommended to use the Connection Pool provided by Middleware (WAS) when the Connection object is frequently created and deleted. This is because the cost of connecting and terminating a Connection is relatively higher than other operations.

5 Error Messages

This chapter lists the SQL States of errors which can occur while using the Altibase JDBC driver.

5.1 SQL States

The string value returned to SQLSTATE is composed of the first 2 characters which indicate the class and the following 3 characters which indicate the subclass. The class indicates the state and the subclass indicates the detailed state.

The following table lists the types of SQLSTATE which can occur in the Altibase JDBC driver and its meanings briefly.

Condition	Class	Subcondition	Subclass
connection exception	08		
		Communication link failure	S01
		Invalid packet header version	P01
		Fail-Over completed	F01
		Invalid format for alternate servers	F02
		Invalid packet next header type	P02
		Invalid packet sequence number	P03
		Invalid packet serial number	P04
		Invalid packet module ID	P05
		Invalid packet module version	P06
		Invalid operation protocol	P07
		Invalid property id: %s	P08
		Invalid connection URL	U01
		Unknown host	H01
		There are no available data source configurations	D01
		connection failure	006

Condition	Class	Subcondition	Subclass
		SQL-client unable to establish SQL-connection	001
		Unsupported Algorithm	K01
		Could not create keystore instance	K02
		Could not load keystore	K03
		Invalid kestore url	K04
		Could not open keystore file	K05
		Key management exception occurred	K06
		Could not retreive key from keystore	K07
		Default algorithm definition invalid	K08
		Mandatory properties that are supported for the client version are not supported for the server version.	M01
dynamic SQL error	07	This statement returns result set(s)	R01
		Invalid query string	Q01
		Statement has not been executed yet	S01
no data	02		
		The sql statement does not produce a result set	001
warning	01		
		cursor operation conflict	001
		Invalid connection string attribute	S00
		Batch update exception occurred: %s	B00
		There are no batch jobs	B01
		There are existing some batch jobs	B02

Condition	Class	Subcondition	Subclass
		The query cannot be executed while batch jobs are executing	B03
		Binding cannot be permitted during executing batch jobs	B04
		Fetch operation cannot be executed during batch update	B05
		There are too many added batch jobs	B31
		Statement has already been closed	C01
		The result set has already been closed	C02
		The stream has already been closed	C03
		additional result sets returned	00D
		This result set doesn't retain data	R01
		Attempt to return too many rows in only one fetch operation	R02
		Option value changed	S02
		Invalid value for bitset	V01
feature not supported	0A		
		Cannot change the name of the database	C01
		The read only mode in transaction cannot be supported	C02
		Not supported operation on forward only mode	T01
		Not supported operation on read only mode violate the JDBC specification	T02
		violate the JDBC specification	V01
syntax error or access rule violation	42	Invalid type conversion	001

Condition	Class	Subcondition	Subclass
		Column not found	S22
JDBC internal error	JI		000
		Overflow occured on dynamic array which is defined by JDBC	D01
		Underflow occured on dynamic array which is defined by JDBC	D02
		This result set was created by JDBC driver's internal statement	D03
		Connection thread is interrupted	D04
		Remaining data exceeds the max size of the primitive type	D05
		Packet Operation has been twisted	P01
		Invalid method invocation	I01
cardinality violation	21	Insert value list does not match column list	S01
data exception	22		000
		null value not allowed	004
		invalid parameter value	023
		Insufficient number of parameters	P01
		IN type parameter needed	P02
		OUT type parameter needed	P03
		There is no column which needs to bind parameter.	P04
		Statement ID mismatch	V01
		Error occured from InputStream	S01

Condition	Class	Subcondition	Subclass
		The length between actual lob data and written lob data into the communication buffer is different.	L01
invalid transaction state	25		
		branch transaction already active	002
savepoint exception	3B		
		Cannot set savepoint at auto-commit mode	S01
		Invalid savepoint name	V01
		Invalid savepoint	V02
		Not supported operation on named savepoint	N01
		Not supported operation on un-named savepoint	N02
invalid schema name	3F		000
		Explain Plan Error	EP
		EXPLAIN PLAN is set to OFF	S01
General Error	HY		
		There are too many allocated statements	000
		Associated statement is not prepared	007
		Attribute cannot be set now	011
		Invalid string or buffer length	090
		Invalid cursor position	109
		Empty ResultSet	R01
		Timeout expired	T00

Condition	Class	Subcondition	Subclass
XA error	XA		
		XA open failed	F01
		XA close failed	F02
		XA recover failed	F03

Appendix A. Data Type Mapping

This appendix lists the compatibility between Altibase data types and standard JDBC data types/Java data types.

Data Type Mapping

The following table shows the basic mapping relationship between JDBC data types, Altibase JDBC data types and Java language types.

JDBC Type	Altibase Type	Java Type
CHAR	CHAR	String
VARCHAR	VARCHAR	String
LONGVARCHAR	VARCHAR	String
NUMERIC	NUMERIC	BigDecimal
DECIMAL	NUMERIC	BigDecimal
BIT	VARBIT	BitSet
BOOLEAN	-	-
TINYINT	SMALLINT	Short
SMALLINT	SMALLINT	Short
INTEGER	INTEGER	Integer
BIGINT	BIGINT	Long
REAL	REAL	Float
FLOAT	FLOAT	BigDecimal
DOUBLE	DOUBLE	Double

JDBC Type	Altibase Type	Java Type
BINARY	BYTE	byte[]
VARBINARY	BLOB	Blob
LONGVARBINARY	BLOB	Blob
DATE	DATE	Timestamp
TIME	DATE	Timestamp
TIMESTAMP	DATE	Timestamp
CLOB	CLOB	Clob
BLOB	BLOB	Blob
ARRAY	-	-
DISTINCT	-	-
STRUCT	-	-
REF	-	-
DATALINK	-	-
JAVA_OBJECT	-	-
NULL	-	null
-	GEOMETRY	byte[]

Converting Java Data Types to Database Data Types

The following table shows the database data types available for conversion for each object when setting an object for a parameter with the setObject method.

	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL/NUMERIC	BIT	CHAR	VARCHAR/LONGVARCHAR	BINARY	VARBINARY/LONGVARBINARY	DATE	TIME	TIMESTAMP	BLOB	CLOB
Array																	
Blob																o	
Boolean	o	o	o	o	o	o	o	o	o	o							
byte[]									o	o	o	o				o	
char[]	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o		o
Clob																	o
Double	o	o	o	o	o	o	o	o	o	o							
Float	o	o	o	o	o	o	o	o	o	o							
Integer	o	o	o	o	o	o	o	o	o	o							
Java class																	
BigDecimal	o	o	o	o	o	o	o	o	o	o							
java.net.URL																	
java.sql.Date									o	o			o	o	o		
java.sql.Time									o	o			o	o	o		
java.sql.Timestamp									o	o			o	o	o		
java.util.BitSet								o									
Long	o	o	o	o	o	o	o	o	o	o							
Ref																	

	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL/NUMERIC	BIT	CHAR	VARCHAR/LONGVARCHAR	BINARY	VARBINARY/LONGVARBINARY	DATE	TIME	TIMESTAMP	BLOB	CLOB
Short		o	o	o	o	o	o	o	o	o							
String	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o		
Struct																	
InputStream																o	
Reader																	o

Converting Database Data Types to Java Data Types

The following table shows whether or not conversion is possible for each database data type with the getXXX method.

	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL/NUMERIC	BIT	CHAR/VARCHAR	LONGVARCHAR	BINARY	VARBINARY/LONGVARBINARY	DATE	TIME	TIMESTAMP	BLOB	CLOB
getArray																	
getAsciiStream	o	o	o	o	o	o	o	o	o	o	o		o	o	o		
getBigDecimal	o	o	o	o	o	o	o		o	o							
getBinaryStream	o		o	o	o	o	o	o			o		o	o	o		

	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL/NUMERIC	BIT	CHAR/VARCHAR	LONGVARCHAR	BINARY	VARBINARY/LONGVARBINARY	DATE	TIME	TIMESTAMP	BLOB	CLOB
getBlob												o				o	
getBoolean	o	o	o	o	o	o	o	o	o	o	o						
getByte	o	o	o	o	o	o	o	o	o	o	o						
getBytes	o	o	o	o	o	o	o	o			o		o	o	o		
getCharacterStream	o	o	o	o	o	o	o	o	o	o	o		o	o	o		
getClob																o	
getDate									o	o			o	o	o		
getDouble	o	o	o	o	o	o	o	o	o	o	o						
getFloat	o	o	o	o	o	o	o	o	o	o	o						
getInt	o	o	o	o	o	o	o	o	o	o	o						
getLong	o	o	o	o	o	o	o	o	o	o	o						
getObject	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
getRef																	
getShort	o	o	o	o	o	o	o	o	o	o	o						
getString	o	o	o	o	o	o	o	o	o	o	o		o	o	o		
getTime									o	o			o	o	o		
getTimestamp									o	o			o	o	o		

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

P

R

S

T

V

W