Cloudscape 3.6's JDBC Driver

A Cloudscape White Paper

Ames Carlson

March 20, 2001

Cloudscape's JDBC driver implements the standard JDBC™ (Java Database Connectivity) interface defined by Sun. For general information about JDBC, consult Sun's Java web site at http://java.sun.com/products/jdbc.

Driver Type

The Cloudscape JDBC driver is a native protocol all-Java driver (Type #4 among the categories defined by JavaSoft).
Cloudscape™ is a pure Java object-relational database management system (ORDBMS) that runs embedded in an application. The application can be a standalone Java application, a client/server application using a server such as the BEA WebLogic Server, the RmiJdbc Server, a servlet engine, or a J2EE engine.
When invoked from a standalone application, the Cloudscape JDBC driver supports connections to Cloudscape databases in embedded mode. In this case, Cloudscape runs in the same JVM as the application and no network transport is required to access the database.
Cloudscape provides an optional connectivity framework, Cloudconnector™, which is a subset of the BEA WebLogic application server. Cloudconnector enables Cloudscape to be used in client/server mode, in which the client application dispatches JDBC requests to the server over a network. When used in this way, Cloudscape can be accessed simultaneously by concurrent clients connecting to the server. Cloudscape can be embedded in other application server frameworks in the same way. Running Cloudscape in such a framework, including one's own, requires a Cloudscape multi-user license, even if multiple client connections are multiplexed to a single server connection.

For a writeup on how Cloudscape is embedded in Cloudconnector, including instructions on how to start and stop Cloudscape with the startup and shutdown of WebLogic 5.1, see our white paper Embedding Cloudscape in WebLogic Server.

Compatibility

Platforms

The Cloudscape JDBC driver has been tested and certified on Windows, Solaris, HP, RS/6000, Novell, Linux, Mac, Psion, and other platforms. However, because the JDBC driver is 100% pure Java, it runs on any platform that supports JDK 1.1 and Java 2. Current JVM validation information is available at http://www.cloudscape.com/support/TechInfo/#JVM.

JDBC Version

The driver is compatible with JDBC version 1.22 and JDBC version 2.0.

JDK Version

The driver requires JDK version 1.1 or higher.

Using the Cloudscape JDBC Driver

To access a Cloudscape database, the application must perform the following tasks:

- Import the JDBC classes
- Register the Cloudscape JDBC driver or application server JDBC client driver
- Establish a connection to the database through the driver

Importing the JDBC Classes

Import the BigDecimal and JDBC classes into the application. To do this, insert the following import statements at the beginning of your program:

```
import java.math.*; //imports the BigDecimal class for JDBC
import java.sql.*;  //imports the JDBC classes
```

Registering the Driver

If Cloudscape is running embedded in the application, use the following call to register the Cloudscape driver with the JDBC Driver Manager:

```
Class.forName("COM.cloudscape.core.JDBCDriver");
```

If Cloudscape is running in client/server mode, register the server's JDBC driver with the JDBC Driver Manager. For example, if you are using Cloudconnector, use the following call to register Cloudscape's client/server JDBC driver in the Cloudscape application:

```
Class.forName("COM.cloudscape.core.WebLogicDriver");
```

An example using RmiJdbc:

```
Class.forName("COM.cloudscape.core.RmiJdbcDriver");
```

Establishing a Connection

Connect to the database using JDBC Driver Manager's *getConnection()* method. The following call creates a database called *mydb* if it does not exist and connects to it through the embedded JDBC driver:

```
DriverManager.getConnection("jdbc:cloudscape:mydb;create=true");
```

The next example creates a database called *mydb* (if it does not already exist) in Cloudconnector and connects to it through the client/server JDBC driver:

```
DriverManager.getConnection("jdbc:cloudscape:weblogic:mydb;create=true");
```

Cloudscape uses BEA WebLogic to provide the Cloudconnector server connectivity.

If Cloudconnector resides on a host that is not local to the client host, it is necessary to specify the hostname and port number. The following example creates and connects to *mydb* on a machine named remoteHost on port 7001:

```
DriverManager.getConnection
  ("jdbc:cloudscape:weblogic://remoteHost:7001/mydb;create=true");
```

The next example creates a database called *mydb* (if it does not already exist) in RmiJdbc and connects to it through the client/server JDBC driver:

```
    DriverManager.getConnection("jdbc:cloudscape:rmi:mydb;create=true");
```

If RmiJdbc resides on a host that is not local to the client host, it is necessary to specify the hostname and port number. The following example creates and connects to *mydb* on a machine named remoteHost on port 1099:

```
    DriverManager.getConnection
        ("jdbc:cloudscape:rmi://remoteHost:1099/mydb;create=true");
```

Minimal Program

The following sample class connects to an embedded Cloudscape database named *mydb*:

```
//import math and jdbc classes
import java.math.*;
import java.sql.*;

public class sample {
    public static void main (String[] args) {
        try {
            //register the driver
            Class.forName("COM.cloudscape.core.JDBCDriver");
            System.out.println("Cloudscape started!");
            //Connect to the database
            Connection conn = DriverManager.getConnection
                 ("jdbc:cloudscape:mydb");
        } catch (Throwable e) {
            System.out.println("exception thrown:"+e);
        }
    }
}
```

Standard JDBC Features

This section summarizes how the Cloudscape JDBC driver handles the following standard features:

- Datatypes
- Stored Procedures
- Prepared Statements
- SQL Batches
- Result Sets
- Database Metadata
- SQL92 Syntax
- Transaction Isolation Level
- Firewall Security

Datatypes

The Cloudscape JDBC driver supports the SQL datatypes required by JDBC 1.22. As an object-relational database, it also supports Java object storage and access within Cloudscape. Java objects stored within Cloudscape can be serialized and streamed back to the application.
The following table shows the mapping between standard JDBC type codes and Cloudscape datatypes.

JDBC Type Code	**Cloudscape Datatype**
Types.BIGINT	LONGINT
Types.BINARY	BIT(numbits)
Types.BIT	BOOLEAN
Types.CHAR	CHAR(numchars)
Types.CHAR	NATIONAL CHAR(numchars)
Types.DATE	DATE
Types.DECIMAL	DECIMAL(precision,scale)
Types.DOUBLE	DOUBLE PRECISION
Types.FLOAT	DOUBLE PRECISION
Types.INTEGER	INTEGER
Types.LONGVARBINARY	LONG BIT VARYING
Types.LONGVARCHAR	LONG NVARCHAR
Types.LONGVARCHAR	LONG VARCHAR
Types.NUMERIC	DECIMAL(precision,scale)
Types.OTHER	Java classes
Types.REAL	REAL
Types.SMALLINT	SMALLINT
Types.TIME	TIME
Types.TIMESTAMP	TIMESTAMP
Types.TINYINT	TINYINT
Types.VARBINARY	BIT VARYING(numbits)
Types.VARCHAR	NATIONAL CHAR VARYING(maxnumchars)
Types.VARCHAR	VARCHAR(maxnumchars)

Cloudscape 3.6 does not support the following JDBC2.0 types:

- java.sql.Ref
- java.sql.Array
- java.sql.Struct

You can store objects and arrays in the database directly as native Java objects, rather than using Ref, Array, and Struct.

Cloudscape 3.6 does not support the following JDBC2.0 java.sql.Type constants:

- JAVA_OBJECT – use OTHER for passing Java objects back and forth with Cloudscape. This is done so that the driver can operate in both JDK1.1 and JDK1.2 environments.
- DISTINCT
- STRUCT
  ARRAY
- BLOB – use LONGVARBINARY for passing blobs back and forth with Cloudscape. This is done so that the driver can operate in both JDK1.1 and JDK1.2 environments.
- CLOB – use LONGVARCHAR for passing blobs back and forth with Cloudscape. This is done so that the driver can operate in both JDK1.1 and JDK1.2 environments.
- REF

Cloudscape uses Java serialization for storing object types, not the new SQLData, SQLInput, or SQLOutput classes.

Cloudscape provides LONG VARCHAR and LONG BIT VARYING types. On the embedded JDBC driver, these can be retrieved using Clob or Blob types. However, on RmiJdbc or Cloudconnector, they can only be retrieved as Java String and binary[] types, not as Clob or Blob types.

Cloudscape's embedded JDBC driver implements Blob and Clob using the existing binary and character types respectively; there are no separate Clob and Blob datatypes in Cloudscape. We recommend using LONG VARBINARY for Blobs and LONG VARCHAR for Clobs, since the sizes of columns with these types is unlimited. However, Blobs will work on types BINARY and VARBINARY as well, while Clobs will work on types CHAR and VARCHAR as well, subject to the size limitations of these types.

Stored Procedures

Most traditional RDBMSs support stored procedures, which are procedural programming language routines that are registered and stored in the database and can be executed from an SQL statement. Because you can invoke the methods of a Java class from within Cloudscape, there is no need for a special category of stored procedures or procedural SQL. You simply create a Java class that implements the procedure and make it available to Cloudscape.

For example, the following SQL VALUES statement invokes a Java method on the class EmailMap, called *emailFromName()*, which returns a person's email address from a table that stores email addresses with other identifying information. We have created a class alias for EmailMap as well.

```
    VALUES EmailMap.emailFromName(GetCurrentConnection(), 'joe', 'smith')
```

Instead of using an SQL stored procedure, the EmailMap class implements the procedure in Java. The EmailMap class must be in the CLASSPATH.

```
public class EmailMap
{
      public static String emailFromName(Connection conn,
            String first, String last)
            PreparedStatement ps = conn.prepareStatement(
                  "SELECT email FROM person WHERE last = ?"+
                  "AND first = ?");
            ps.setString(1, last);
            ps.setString(2, first);
            Result rs = ps.executeQuery();
            String email = (rs.next())? rs.getString(1) : "unknown";
            rs.close();
            ps.close();
            return email;
      }
}
```

Cloudscape JDBC driver supports INOUT and OUT parameters on its Java stored procedures. To use INOUT or OUT parameters, you first define the Java method to take an array of the desired parameter type, and then use JDBC's CallableStatement methods to access the output value.

For example, we can redo the above method to pass the value back in a parameter rather than as a result of the method call:

```
public class EmailMap
{
      public static void emailFromName2(
            String first, String last, String[] email)
            Connection conn =DriverManager.getConnection(
                "jdbc:default:connection");
            PreparedStatement ps = conn.prepareStatement(
                  "SELECT email FROM person WHERE last = ? "+
                  "AND first = ?");
            ps.setString(1, last);
            ps.setString(2, first);
            Result rs = ps.executeQuery();
            String emailValue = (rs.next())? rs.getString(1) : "unknown";
            rs.close();
            ps.close();
            conn.close();
            email[0] = emailValue;
      }
}
```

And then invoke the method and get the value out like so:

```
      CallableStatement cs = conn.prepareCall(
            "CALL EmailMap.emailFromName2(?,?,?)");
      cs.setString(1, 'joe');
```

```
cs.setString(2, 'smith');
cs.registerOutParameter(3, java.sql.Types.VARCHAR);
cs.executeUpdate();
String email = cs.getString(3);
```

Users can also retrieve the return value of a method as an OUT parameter using the `"? = CALL ..."` syntax:

```
CallableStatement cs = conn.prepareCall(
      "? = CALL EmailMap.emailFromName(GetCurrentConnection(),?,?)");
cs.setString(1, 'joe');
cs.setString(2, 'smith');
cs.registerOutParameter(1, java.sql.Types.VARCHAR);
cs.executeUpdate();
String email = cs.getString(1);
```

Prepared Statements

In addition to supporting standard dynamic SQL, where you construct and compile your SQL statement at run time, Cloudscape supports precompiling statements and saving them for re-use. This Cloudscape feature is called Stored Prepared Statements.

Prior to running your application, you would identify those statements that can be precompiled. These statements need to be known ahead of time; they can contain parameters that are set at run time. You would then create these statements once, and use them from any appropriate applications. The same stored prepared statement can be used by several applications.

Here is an example DDL statement showing creation of a Stored Prepared Statement with a parameter:

```
CREATE STORED STATEMENT InsertEmail AS
INSERT INTO person (first, last, email) VALUES (?,?,?)
```

This statement can then be used from a JDBC application, supplying values for the parameters like so:

```
PreparedStatement ps = conn.prepareStatement("CALL InsertEmail");
ps.setString(1,"jill");
ps.setString(2,"java");
ps.setString(3,"jill.java@javau.edu");
ps.executeUpdate();
```

You can make almost any statement a stored prepared statement, including SELECT statements that return ResultSets for further processing. More information about Stored Prepared Statements are available in the Cloudscape User Documentation.

Stored Prepared Statements will be recompiled automatically if they are impacted by changes to the database's schema or statistics; commands are available for manual recompilation as well.

SQL Batches

The Cloudscape JDBC driver supports JDBC 2.0 SQL batches. You can either batch multiple SQL statements using a java.sql Statement object, or batch multiple calls of a single SQL statement using a java.sql.PreparedStatement object.

This example batches multiple SQL statements:

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO person
    VALUES('jill','java','jj@whoozit.com') ");
stmt.addBatch("UPDATE friends SET numFriends = numFriends+1 "+
    "WHERE first='joe' AND last='smith'");

int [] updateCounts = stmt.executeBatch();
con.commit();
```

This example batches multiple calls of a single SQL statement using a PreparedStatement:

```
con.setAutoCommit(false);
PreparedStatement ps = con.prepareStatement(
      "INSERT INTO person VALUES (?, ?, ?)");

ps.setString(1,"jack");
ps.setString(2,"jones");
ps.setString(3,"jack@whatzit.com");
ps.addBatch();

ps.setString(1,"sally");
ps.setString(2,"smith");
ps.setString(3,"ssmith@newco.com");
ps.addBatch();

int [] updateCounts = ps.executeBatch();
con.commit();
```

Result Sets

The Cloudscape JDBC driver supports JDBC 1.2 Result Set processing and JDBC 2.0 scrolling insensitive cursors.

This example creates a scrolling insensitive cursor:

```
Statement stmt = con.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery(
                 "SELECT email, first, last FROM person");
```

Once you have this scrollable result set, you can scroll forward, backward, and to arbitrary locations using the new calls on java.sql.ResultSet. For example:

```
rs.first(); // goes to the first row

rs.absolute(5); // goes to the fifth row

rs.relative(-2); // backs up two rows

rs.last(); // goes to the last row

rs.previous(); // backs up one row
```

It is recommended that when using scrolling ResultSets, you turn autocommit off. When autocommit is on, moving to the last row of the result set with a next() call will cause it to close. This can be done with the call:

```
con.setAutoCommit(false);
```

ResultSets are closed when commits are performed, so a commit should only be issued once processing of a ResultSet has completed.

Database Metadata

The Cloudscape JDBC driver supports all the standard JDBC 1.2 and 2.0 DatabaseMetaData methods. For those features not yet implemented in Cloudscape, such as privileges, empty result sets of the required shape are returned. This includes:

- getColumnPrivileges
- getTablePrivileges

SQL92 Syntax

Cloudscape supports SQL 92 Entry and significant portions of SQL 92 Transitional and Intermediate. It also supplies standard SQL-J extensions to SQL. The Cloudscape JDBC driver supports all JDBC escape clauses:

- date, time, and timestamp escape clauses
- outer join escape clause
- call escape clause
- like escape's escape clause
- function escape clause, including support for all of the Cloudscape builtin functions including ABSOLUTE, LOCATE, SQRT, SUBSTRING, SUBSTR, and also the following functions: ABS, CONCAT, LENGTH, CURDATE, CURRENT_DATE, CURTIME, CURRENT_TIME.

Transaction Isolation Level

The Cloudscape JDBC driver supports the Serializable (Level 3), Repeatable Read (Level 2), and Read Committed (Level 1) isolation levels. Requests for Read Uncommitted (Level 0) isolation set isolation to Read Committed. The default isolation level for a new connection is Read Committed.

Firewall Security

The Cloudscape JDBC driver cannot directly connect to a database behind a firewall, because the firewall prevents the browser from opening a TCP/IP socket to the database.

If you are using Cloudconnector, you can use HTTP tunneling to access a database behind a firewall. See Cloudconnector's WebLogic documentation for details on HTTP tunneling.

JDBC 2.0 Optional Package

JDBC 2.0 has optional packages, formerly referred to as standard extensions. This section summarizes the following JDBC 2.0 optional packages supported by Cloudscape:

- Data Source
- Connection Pool
- Distributed Transactions

Data Source

Cloudscape implements the JDBC 2.0 Data Source optional package. This extension allows the application developer to develop portable applications by removing database-specific JDBC driver and JDBC URL information from the application. This information is instead recorded within a DataSource object located via JNDI, typically in an LDAP server.

To create a Cloudscape DataSource, you register the DataSource object with a JNDI server. This is an administrative task that is data-source specific. This example gets a Cloudscape Data Source object:

```
import COM.cloudscape.core.*;
import javax.sql.DataSource;

// This method uses the Cloudscape classes BasicDataSource and
// DataSourceFactory to register a Cloudscape DataSource object.
DataSource makeDataSource(String databaseName, String machine,
    int port) throws Exception
{
      BasicDataSource ds = DataSourceFactory.getDataSource();
      ds.setDatabaseName(databaseName);
      if (machine != null) // null = embedded data source
      {
            ds.setRemoteDataSourceProtocol(
                "rmi://"+machine+":"+port+"/");
      }
      return ds;
}
```

Code such as the following would create and register a Cloudscape Data Source object with a JNDI server:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/MyDB", makeDataSource("MyDB","myHost",1099));
```

To use a Cloudscape Data Source, you use a JNDI Context and look up the DataSource object using the name supplied to you by the administrator. For example:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/MyDB");
Connection con = ds.getConnection("myPassword", "myUserName");
```

Connection Pool

Cloudscape implements the JDBC 2.0 Connection Pool optional package.

To create Connection Pool DataSource objects, administrators use the getConnectionPoolDataSource() method on COM.cloudscape.core.DataSourceFactory. This will return a Connection Pool DataSource object, which can then be registered and retrieved like other DataSource objects.

Cloudscape also supplies an implementation of this for the RmiJdbc server framework.

Cloudconnector 3.6 does not support the Connection Pool optional package; instead, it supplies its own connection pooling, which is described later in this paper.

Distributed Transactions

Cloudscape implements the JDBC 2.0 distributed transactions optional package. This optional package allows more than one database or connection to participate in the same transaction.

Distributed transactions use XADataSource entries rather than DataSource or ConnectionPoolDataSource entries. Such entries can participate in distributed transactions using the methods provided in the XADataSource implementation. See the JTA specifications for more information.

To create XADataSource objects, administrators use the getXADataSource() method on COM.cloudscape.core.DataSourceFactory. This will return an XADataSource object, which can then be registered and retrieved like other DataSource objects.

Cloudscape also supplies an implementation of this for the RmiJdbc server framework; use the getRemoteXADataSource() method on COM.cloudscape.core.DataSourceFactory to get an XADataSource object over RmiJdbc.

Cloudconnector 3.6 does not support the distributed transactions optional package.

Cloudscape JDBC Extensions

This section summarizes the following Cloudscape JDBC extensions:

- Row Prefetching and Caching
- Nested Connections
- Mulithreaded/Shared Connections

Row Prefetching and Caching for Server-Side JDBC

Row prefetching and caching is supported when a Cloudscape database is accessed through Cloudconnector. Prefetching and cacheing can improve performance by reducing the number of round trips between the client and the database when data rows are returned. The trade-off for the reduced network traffic is the requirement of more memory for the cache.

Cloudconnector's default behavior is to prefetch and cache 25 rows per round trip between the client and Cloudscape. You can tune this behavior by setting the *cacheRows* property to a greater or lesser number of rows. You can set the property in the connect URL, as in the following example, which increases the number of cached rows to 100 for the duration of the connection:

```
DriverManager.getConnection(
"jdbc:cloudscape:weblogic:mydb;create=true&weblogic.t3.cacheRows=100");
```

You can also set the *cacheRows* property for a particular operation, and then reset it when the operation is complete. For example, to support a query on an updatable cursor you would need to turn off prefetching and caching by setting cacheRows to zero. You could then reset it after the query executes.

```
//turn off prefetch and caching
weblogic.t3.cacheRows=0;

// updatable cursor operation goes here

//reset prefetch and caching to the default
weblogic.t3.cacheRows=25;
```

Nested Connections

Cloudscape permits server-side JDBC calls through nested connections. A nested connection shares the transaction control and lock space of its parent connection. This can be used to ensure that a specific operation, for example an UPDATE that executes inside a SELECT, is atomic. An application can create a nested connection in three different ways:

Getting the current connection with the built-in Cloudscape method, *GetCurrentConnection()*, and then passing the connection to the method that is being nested:

```
SELECT City.getDistanceFrom(GetCurrentConnection(), 'Paris', 'France')
FROM Cities WHERE city_id = 14;
```

Within the nested method, establishing the connection with the "current=true" property instructs the driver to re-use the current connection:

```
DriverManager.getConnection("jdbc:cloudscape:mydb;current=true");
```

Establishing the connection with the "jdbc:default:connection" JDBC URL instructs the driver to re-use the current connection:

```
DriverManager.getConnection("jdbc:default:connection");
```

Multithreaded/Shared Connections

Multiple threads may safely share the same Cloudscape connection. However, because Cloudscape synchronizes JDBC requests, only one request is executed at a time on a single connection.

If you are assigning multiple threads to a single connection, you need to take precautions to ensure that one thread does not affect the behavior of another. Be aware of the following behavior when coding a JDBC connection with multiple threads:

- When a thread commits, it closes the statements and result sets of all the other threads on the same connection.
- If multiple threads share statements, they may close each other's result sets.

This is standard JDBC API behavior.

Applets

An applet cannot load and access Cloudscape embedded in the browser's JVM because of the Java sandbox security model, which prevents an applet from writing to disk , unless the applet is signed. To connect to Cloudscape, an applet must connect either to a servlet or to an application server framework running Cloudscape, such as Cloudconnector or J2EE.

Usually, the applet is downloaded from a Web server and, when loaded on the client (Browser), connects to Cloudconnector. Because of applet security restrictions, the applet can only connect to Cloudconnector when it is running on the same host as the Web server where it was originally downloaded.

To optimize applet download time, various classes composing the client-side JDBC driver can be packaged in a .jar file, so that the download is performed in one request rather than a class at a time.

In a JDK 1.1.1- based web browser, such as Netscape 4.0, an applet can request socket connection privileges and, if granted them, the applet can connect to a database running on a different host from the web server host that it is connecting to.
In Netscape 4.0, this involves signing the applet, then opening the connection as follows:

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
connection = DriverManager.getConnection
        ("jdbc:cloudscape:weblogic://remoteHost:7001/mydb");
```

Refer to your browser documentation for detailed information on how to do this.

Servlets and JSPs

Cloudscape can run inside a servlets or JSP and be accessed remotely via the HTTP protocol. Cloudconnector, Sun's Java Web Server, Apache, and many other web servers and application servers support servlets. Loading the embedded Cloudscape JDBC driver in the *init()* method of the servlet causes Cloudscape to listen for requests. You can use techniques such as database connection pools to maintain opened connection to Cloudscape. Cloudconnector supports database connection pool functionality. Then, the servlet's *service()* method can be implemented or overridden to access and send statements to Cloudscape as well as to process and return results to the client.

Refer to your web server or application server documentation for detailed information on how to configure and enable servlets. Cloudscape provides a simple servlet example for Cloudconnector and Java Web Server in the sample application classes for the tutorial that come with the product. White papers and technical notes showing how to use Cloudscape within Apache/JServ, Enhydra and Tomcat are also available on the Cloudscape web site.

For More Information

For more information on developing applications using the Cloudscape JDBC driver, see the Cloudscape Developer's Guide. Any questions or comments about this white paper should be sent to cloudhelp@informix.com.

Links used in this document

Sun's JDBC web site	http://java.sun.com/products/jdbc
BEA WebLogic Server	http://weblogic.beasys.com/products/weblogic/server/index.shtml
RmiJdbc Server	http://www.objectweb.org/RmiJdbc/
Embedding Cloudscape in WebLogic Server	http://www.cloudscape.com/pdf/EmbedCloudscapeWebLogic.pdf
Cloudscape JVM validation information	http://www.cloudscape.com/support/TechInfo/#JVM
JDBC 2.0 Optional Package	http://java.sun.com/j2se/1.3/docs/guide/jdbc/index.html
JTA specifications	http://java.sun.com/products/jta/index.html
Signing Netscape applets	http://developer.netscape.com/docs/manuals/signedobj/capabilities/contents.htm
Cloudscape Developer's Guide	http://www.cloudscape.com/support/doc_36/doc/html/coredocs/dgtit.htm

With its combination of robust SQL features, support for Java, and embeddable, pure Java architecture, Cloudscape is the data management product of choice for data-driven Java applications.