Cloudscape:
A Technical Overview

A Cloudscape White Paper

Nat Wyatt, Ames Carlson

March 2001

Contents

Introduction

This paper describes some of the technical features of Cloudscape, the embeddable SQL database engine for Java. Because of its lightweight, pure-Java, embeddable architecture, Cloudscape is the ideal database engine for deploying database-driven Java applications. The database engine becomes part of the application, and the user never has to install or manage it – the database becomes invisible.

Cloudscape is a lightweight, embeddable object-relational database engine in the form of a Java class library. Its native interface is JDBC, with "Java-relational" object extensions. It implements the SQL92E standard plus object extensions. The object extensions include the ability to create columns whose type is any Java class or interface, access to the public fields and methods of those objects, some additional Java-SQL syntax (new, instanceof, etc.), Java stored procedures, and the ability load Java classes from the database. The engine provides transactions and crash recovery, and allows multiple connections and multiple threads to use a connection. Because it is a 100% Pure Java class library, Cloudscape can be easily embedded into any Java application program or server framework without compromising the "Java-ness" of the application.

Cloudsync is an add-on option to Cloudscape that provides single-master update-anywhere schema and application synchronization. The Cloudsync protocol synchronizes data, schema, and applications, and guarantees agreement and consistency between all participating databases without intervention at the slave nodes. It provides a mechanism for application-level conflict resolution, and is capable of replicating objects along with their methods. Cloudsync is not covered in detail here; see the *Cloudsync Technical Overview* for details.

Cloudconnector is an add-on option to Cloudscape that provides high-performance, secure client/server JDBC connectivity. Cloudconnector's JDBC functionality includes connection pooling, prefetching, and JDBC over SSL. Cloudconnector is not covered in detail here; see the Cloudscape *Server and Administration Guide* for details.

The technical aspects of Cloudscape which most differentiate it from other database systems are:

- It is itself a 100% pure Java class library. This is important to Java developers who are trying to maintain the advantages of Java, such as platform independence, ease of configuration, and ease of installation.

- It needs no proprietary Java Virtual Machine (JVM). Being 100% pure Java it runs with any certified JVM.

- The Database Management System (DBMS) engine is lightweight: about 2MB of class files, and it uses as little as 4MB of Java heap.

- Cloudscape is embeddable: applications can embed the DBMS engine in the application process, eliminating the need to manage a separate database process or service. Cloudscape can also run as a separate process, using the RmiJdbc or Cloudconnector server framework or your own server framework.

- ❑ It is easy to administer: when embedded in a client application, a Cloudscape system requires no administrative intervention.

- ❑ Cloudscape provides the ability to write stored procedures in Java that can run in any tier of the application. Cloudscape does not have a proprietary stored procedure language.

- ❑ Cloudscape can store and manage Java objects in the database, and access their fields and methods from SQL queries. Any Java class can serve as a SQL datatype and be used to create a column. It is not necessary to preprocess the classes, nor is it necessary for them to derive from a proprietary base class. Cloudscape can also store and load the classes associated with Java data types and stored procedures from the database.

- ❑ Object synchronization capability is built-in.

In other respects, Cloudscape is very much like other relational database systems:

- ❑ It implements the SQL92E language standard and the JDBC API standard.

- ❑ It has transactions (commit and rollback), supports multiple connections with transactional isolation, and provides crash recovery. It allows multiple threads to share the same connection.

This unique combination of technical capabilities allows application developers to build data-driven applications that are pervasive (run anywhere), deployable (downloadable), manageable, extensible, and connectable. It is these technical features that are the subject of the remainder of the paper, which describes:

- General Architecture
  - Database engine as an embeddable class library
  - Programmer's API
  - JDBC Drivers
  - ODBC Driver
  - Relational Capabilities
- Java Extensions
  - Storing Java Objects in Tables
  - Java User-Defined Datatypes
  - Java Stored Procedures
  - Using Java methods in constraints
  - Loading Classes from the Database
  - External Virtual Tables
  - User-defined Aggregates
  - Storing applications in the database
- Security Features

General Architecture

Cloudscape is an object-relational database management engine implemented as a Java class library. Applications access data managed by the Cloudscape engine via the JDBC (Java Database Connectivity) API.

The database engine is embeddable. This means that rather than running as a separate process, the database engine software can be part of the application so that the application and the database engine run in the same Java virtual machine (JVM). Even when embedded, the application uses the JDBC API to access the database, but the embedded JDBC driver simply transfers data to and from the database engine without the need for network communication. Whether or not it is embedded, the database engine supports multiple simultaneous connections and access from multiple application threads.

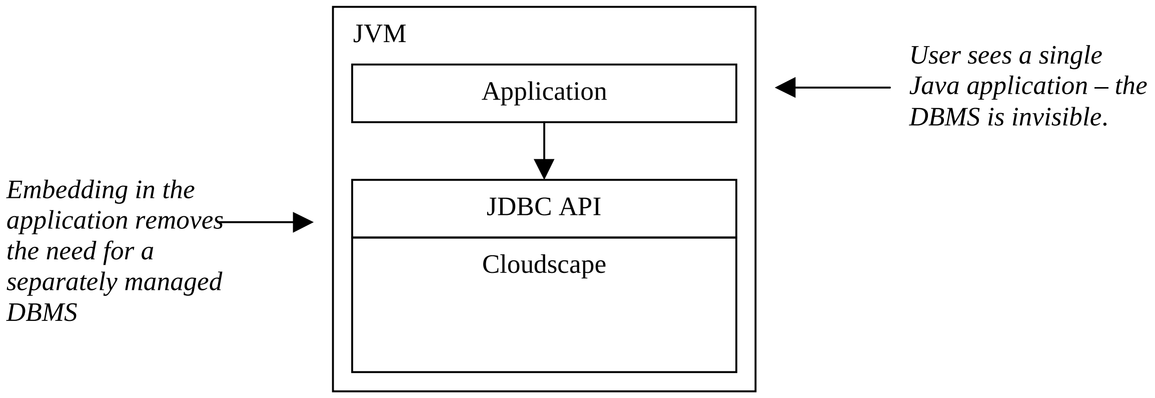
JVM
Application
JDBC API
Cloudscape
User sees a single Java application – the DBMS is invisible.
Embedding in the application removes the need for a separately managed DBMS

Figure 1. Cloudscape Embedded in an Application

It is also possible to deploy Cloudscape as a traditional client/server database server. To accomplish this, Cloudscape is embedded in a *server framework*. A server framework is simply a piece of software that can accept and process network communication. Examples are the Java Web Server for the HTTP protocol, CORBA ORBs (Object Request Brokers), and products such as BEA's Weblogic application server framework Cloudscape uses BEA's Weblogic server as part of the Cloudconnector Server for remote JDBC access.

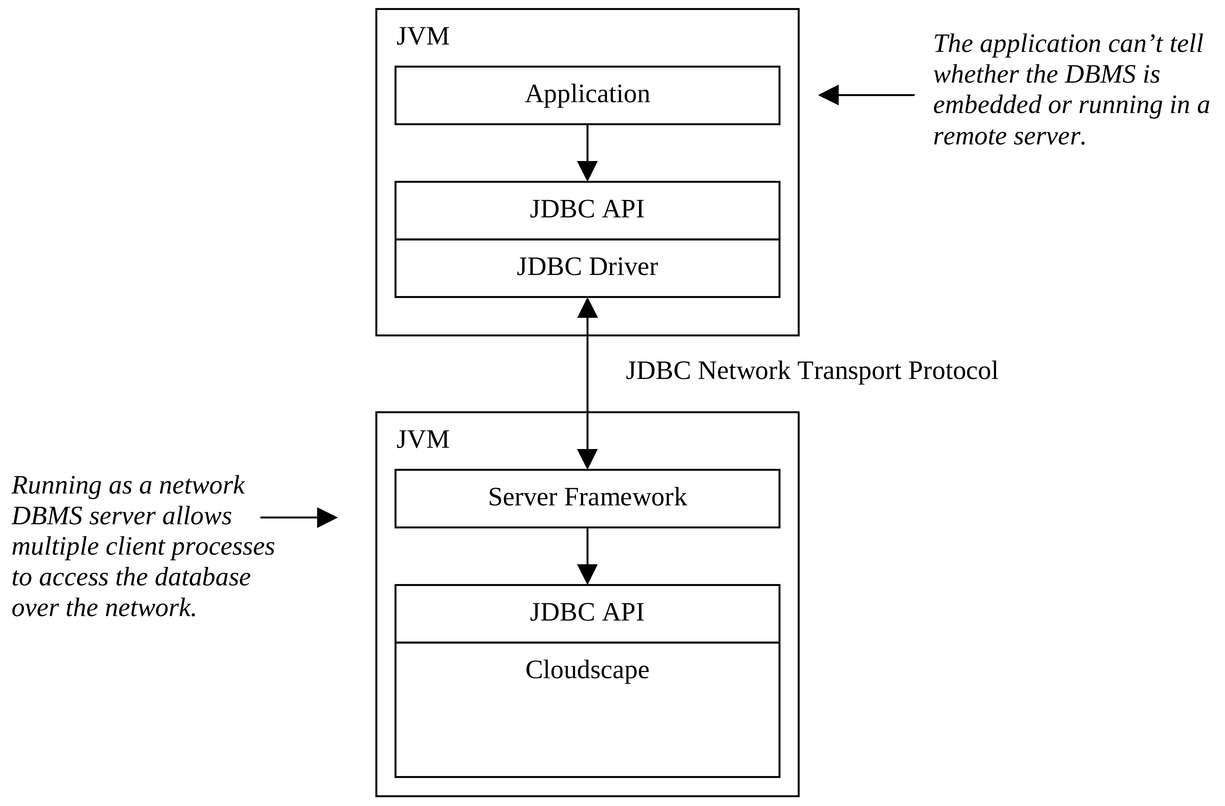
JVM
Application
JDBC API
JDBC Driver
The application can't tell whether the DBMS is embedded or running in a remote server.
JDBC Network Transport Protocol
JVM
Server Framework
JDBC API
Cloudscape
Running as a network DBMS server allows multiple client processes to access the database over the network.

Figure 2. Using a Server Framework, Cloudscape Acts as a Client/Server DBMS

It is just as easy to embed Cloudscape inside a Java web server. Figure 3 shows Cloudscape embedded in a Java web server, with clients accessing the database via HTTP (Hypertext Transmission Protocol) requests to servlets.

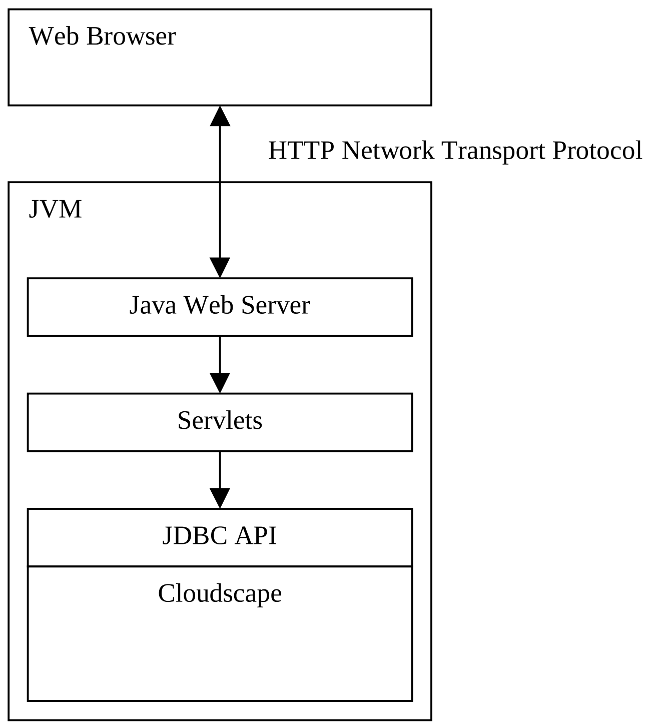

Figure 3. Cloudscape Embedded in Web Server

Cloudscape provides persistence of data by storing data in disk files. A Cloudscape engine can manage one or more database files, but each database file can only, be accessed by a single Cloudscape engine. In a client/server configuration, the engine provides multi-user access to the databases under its control. All threads that access the database do so via the database engine.

.

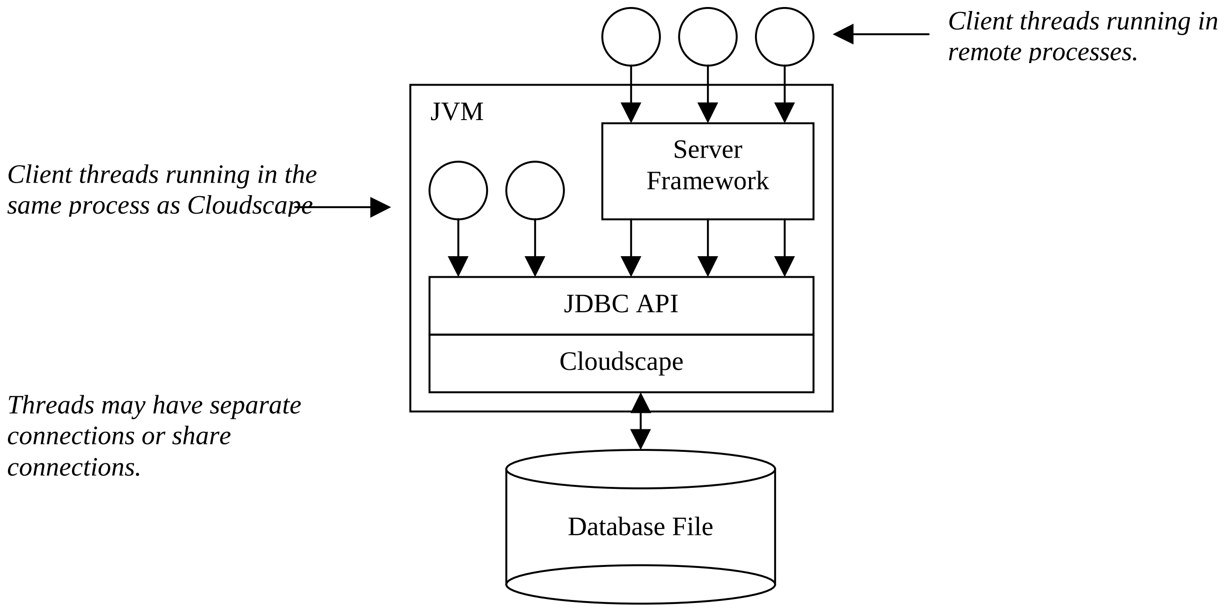
Client threads running in remote processes.
JVM
Server Framework
Client threads running in the same process as Cloudscape
JDBC API
Cloudscape
Threads may have separate connections or share connections.
Database File

Figure 4. Thread Access to Cloudscape

The data in the database disk files is stored in a portable format so that databases can be easily transported from machine to machine regardless of the CPU architecture of the machines. Cloudscape can also handle data files on read-only media. These characteristics make it easy to download Cloudscape applications together with their database, or run them from a CD-ROM. The combination of portable database formats and the 100% Pure Java DBMS engine makes it possible to send a data-centric application anywhere, either on media or over a network.

Cloudscape provides a great deal of flexibility for system designers. Each Cloudscape instance can manage multiple databases, the databases can live on various media, and there's nothing to stop the application from connecting to other DBMS systems.

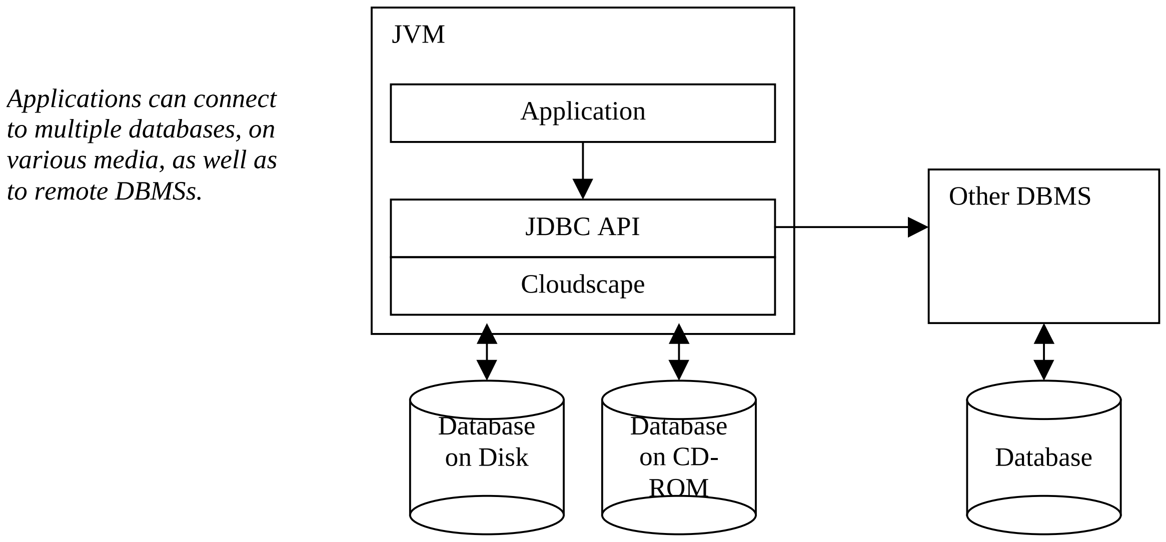
JVM
Application
JDBC API
Cloudscape
Other DBMS
Database on Disk
Database on CD-ROM
Database
Applications can connect to multiple databases, on various media, as well as to remote DBMSs.

Figure 5. Flexible Application Architecture

Programmer's API

Applications access the database with the industry standard JDBC API. When a client application wishes to store or retrieve data from the database, it submits a request via the JDBC API to the Cloudscape engine (either over a network or directly to the embedded engine). The client program is not required to use any Cloudscape-specific APIs; clients use only JDBC. The Cloudscape driver is selected by supplying a Cloudscape JDBC connection URL (Universal Resource Locator). For example,

```
Connection conn = DriverManager.getConnection("jdbc:cloudscape:greetdb");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT message FROM greetings");
while (rs.next())
{
    String greeting = rs.getString(1);
    System.out.println(greeting);
}
rs.close();
stmt.close();
conn.close();
```

Since the application is only using JDBC APIs, it is portable, and will run against JDBC compliant databases from other vendors. This standards compliance enables a developer to write an application against Cloudscape and deploy it against another DBMS.

When a client requests data from the DBMS, the JDBC driver copies the data from the database into the application. Modifications of that data in the application, intentional or unintentional, do not affect the data in the database until JDBC calls are used to send the changes back to the DBMS. Similarly, when a client stores data in the DBMS, it is copied out of the client and into the DBMS. Because Cloudscape is transactional, complex data manipulations can be grouped together into transactions. Cloudscape guarantees data consistency by ensuring that even after system failure, either all or none of the transaction will commit to the database file.

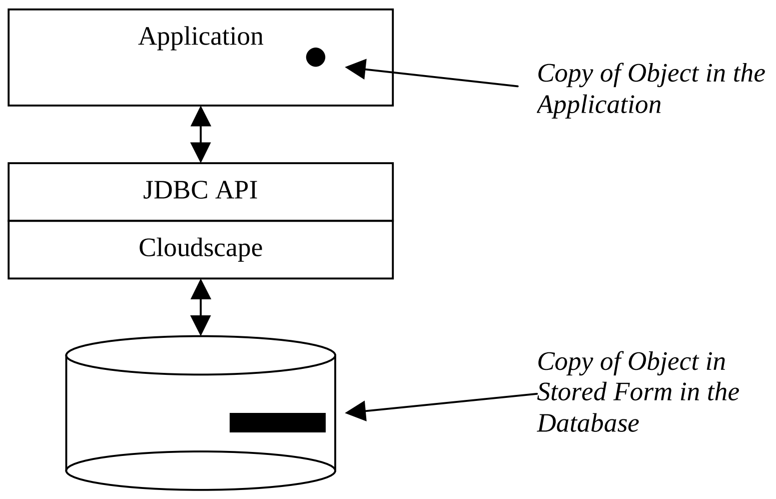
Application
Copy of Object in the Application
JDBC API
Cloudscape
Copy of Object in Stored Form in the Database

Figure 6. Objects Copied from DBMS to Application

More information about JDBC can be found at http://java.sun.com/jdbc.

JDBC Drivers

Cloudscape supplies three drivers for accessing Cloudscape from Java Client programs. These are all *Type 4* drivers: pure Java on the client and connect directly to the database engine without any intermediate translation. The *embedded* driver is used to communicate with Cloudscape when it is running in the same JVM as the application program. The *RmiJdbc* driver communicates with Cloudscape when it's running in the RmiJdbc server that comes with Cloudscape. Cloudconnector's *Weblogic* driver communicates with Cloudscape when it's running in the *BEA Weblogic* application server.

JDBC connection URLs to Cloudscape databases all start with the prefix `jdbc:cloudscape:`. The connection URL for an embedded database on drive e: might look like this: `jdbc:cloudscape:e:/db`. The connection URL to a Cloudscape server might look like this: `jdbc:cloudscape:weblogic://localhost:7777/db`.

Cloudscape JDBC drivers support JDBC2.0; they work with either JDK1.1 or Java 2 JVMs. The standard extensions for JNDI data sources, connection pooling, and XA are also provided. These features enable Cloudscape to be a resource manager in a distributed J2EE system. More information on Cloudscape's JDBC driver can be found in the white paper *Cloudscape JDBC Driver*.

ODBC Driver

Cloudscape also supplies an ODBC 3.0 driver so that non-Java Windows programs can access Cloudscape databases. This makes Cloudscape databases available to a variety of existing programs, such as spreadsheets and report writers. It is possible to access both embedded and remote databases with the ODBC driver.

RDBMS Capabilities

Cloudscape implements most of Core SQL99 any many other features of SQL99, with extensions for Java. SQL language support includes the following:

- Basic types: CHAR, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT.
- Datetime data types (from SQL92T): DATE, TIME, TIMESTAMP (with JDBC date/time escape syntax).
- Other types: BIT and BIT VARYING (from SQL92F), BOOLEAN, LONGINT and TINYINT (from JDBC), VARCHAR (from SQL92T), NCHAR and NVARCHAR (from SQL92F), LONG VARCHAR, LONG VARBINARY, and LONG NVARCHAR (from JDBC).
- Basic math operations: +,*,-,/,unary +,unary -.
- Basic comparisons: <,>,<=,>=,<>,=.
- Datetime literals
- Built-in functions: BIT_LENGTH, CAST, CHAR_LENGTH, concatenation (||), CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, EXTRACT, OCTET_LENGTH, LOWER, SUBSTRING, SUBSTR, LTRIM, RTRIM, UPPER, CURRENT_USER, USER, SESSION_USER, SQRT, and others.
- Basic predicates: BETWEEN, LIKE, NULL
- Quantified predicates: IN, ALL/SOME, EXISTS
- CREATE and DROP SCHEMA
- CREATE and DROP TABLE
- Check constraints
- ALTER TABLE: ADD COLUMN and ADD/DROP CONSTRAINT
- CREATE and DROP VIEW
- Constraints: NOT NULL, UNIQUE, PRIMARY KEY, CHECK, FOREIGN KEY
- Column defaults
- Delimited identifiers
- Updatable cursors (via JDBC)
- Dynamic SQL (via JDBC)
- INSERT, UPDATE, and DELETE statements

- Positioned updates and deletes
- WHERE qualifications
- GROUP BY
- HAVING
- ORDER BY
- UNION and UNION ALL
- Subqueries as expressions (from SQL92F)
- Joins in the WHERE clause
- Joins (SQL92T): INNER, RIGHT OUTER, LEFT OUTER, CROSS, named column join, condition join
- Aggregate functions (with DISTINCT): AVG, COUNT, MAX, MIN, SUM
- Select expressions
- SELECT *, SELECT table.* (SQL92T), SELECT DISTINCT, select expressions
- Named select columns
- SQLSTATE
- UNION in views (SQL92T)
- Implicit numeric casting (SQL92T) and implicit character string casting
- CAST (SQL92T)
- INSERT expressions (SQL92T): *`insert into T2 (COL) select col from T1`*
- VALUES expressions: *`select * from (values (1, 2)) as foo(x, y)`*, etc.
- SET TRANSACTION ISOLATION LEVEL to READ_COMMITTED or SERIALIZABLE
- Triggers

Other traditional database features include:

- Cost-based query optimizer: join order, index selection, bulk fetching, join strategies (nested loop or hash), sort avoidance, lock escalation, subquery flattening, transitive closure, and many other query transformations. It uses a unique sampling technique that requires no intervention for statistical gathering, and also provides query plan overrides and statistics on actual query results.
- Multicolumn B-Tree indexes
- Unlimited length rows, and the capability of "streaming" column values to the client
- Data import/export with fast load and fast create index
- Transaction commit and rollback
- Transactional isolation (serializable, repeatable read, or read committed)
- Crash recovery
- Multiple concurrent connections
- Multithreaded connections
- Row locking with escalation to table locks
- User and connection authentication: built-in SH3, application-defined, or via external user authentication systems (LDAP, NIS+, JNDI)
- Diagnostics and consistency checks
- Online database backup
- Locales
- Database encryption
- Static SQL – precompiled statements

- GUI for schema management, Cloudview
- Script tool, ij
- Modify column to change its length or nullability
- Compress table to recover unused disk space for a table
- Autoincrement default for automatic sequence numbers for data

Java Extensions

Java extensions include the following (these features are described in more detail below):

- The ability to use any Java class or interface as a column type, and to order and index such types.
- Access to public methods and fields in queries.
- Java stored procedures, constraints, and triggers.
- Java virtual tables: the ability to provide and manipulate table data from an arbitrary data source.
- The ability to store application classes in the database.
- SQL language extensions for Java: CLASS, INSTANCEOF, NEW, conditional operator (?:), IS NULL for objects.
- Class and method aliases.
- User-defined aggregates.

Storing Java Objects in Tables

In Cloudscape, Java objects are stored in tables as column values. Any Java class can serve as a column data type. An important aspect of the way Cloudscape does this is that it does not require the class to derive from any base class. The only requirement is that the instances stored in the database implement the java.io.Serializable interface. For example, to create a table with a single column to hold a value of class URL,

```
CREATE TABLE websites (url SERIALIZE(java.net.URL))
```

The SERIALIZE keyword tells Cloudscape how to store the object in the table, in this case by the standard Java serialization mechanism. To insert a value into the table,

```
INSERT INTO websites (url)
VALUES (NEW java.net.URL('http://www.cloudscape.com/support'))
```

It’s possible to use public fields and methods in queries. For example, to find all the sites under www.cloudscape.com:

```
SELECT url FROM websites
WHERE url.getHost() = 'www.cloudscape.com'
```

Field and method access is allowed in the from list as well as in qualifications. To get an alphabetical list of protocols used by the web sites:

```
SELECT DISTINCT url.getProtocol() protocol
FROM websites
ORDER BY protocol
```

Fields and methods can be used in UPDATE and DELETE statements as well, of course:

```
UPDATE websites SET url = NULL
WHERE LOWER(url.getHost()) LIKE '%oracle%'
```

It's possible to call static methods, too. For example, to find out the current CLASSPATH, which is where, Cloudscape is loading all the object classes from:

```
VALUES (CLASS java.lang.System).getProperty('java.class.path')
```

When a Java program wishes to retrieve or change an object stored in a column, it uses the JDBC getObject() and setObject() calls on ResultSet and PreparedStatement. For example, here is a snippet to get the URLs under www.cloudscape.com from the websites table:

```
Connection conn = DriverManager.getConnection("jdbc:cloudscape:webdb");
PreparedStatement ps = conn.prepareStatement(
    "SELECT url FROM websites WHERE url.getHost() = ?");
ps.setString(1, "www.cloudscape.com");
ResultSet rs = ps.executeQuery();
while (rs.next())
{
    URL url = (URL) rs.getObject(1);
    System.out.println(url);
}
rs.close();
ps.close();
conn.close();
```

Similarly, when inserting values, PreparedStatement.setObject() is used to pass objects into the Prepared Statement.

Java User-Defined Datatypes

As shown by the preceding examples (which all use existing classes from the Java Developer's Kit API), Cloudscape's approach to providing object storage makes it easy to store any existing Java object in the database. However, the typical use for this capability is to add useful application-specific types to SQL. These are often referred to as *user defined types*, or UDTs. For example, the following class is a simple UDT that maintains address information:

```
import java.io.Serializable;

public class Address implements Serializable, Comparable
{
    public int number;
    public String street;
    public int zipcode;
    public Address(int number, String street, String city, String state)
    {
        this.number = number;
        this.street = street;
        this.zipcode = ZipMap.zipFromCity(city, state);
    }

    public String city()
    {
        return ZipMap.cityFromZip(zipcode);
    }

    public int compareTo(Object o)
    {
        // returns <0, 0, or >0 for LT, EQ, GT
        return zipcode - ((Address) o).zipcode;
    }

    public String toString()
    {
        return number + " " + street + ", " + city() + " " + zipcode;
    }
}
```

This class keeps track of postal address information by storing the street, number, and zip code, and computes the name of the city from the zip code. Cloudscape understands the following things about this class:

- Since it implements the java.io.Serializable interface, Cloudscape can store instances of this class in a column. Note that no explicit coding is required for this; Java provides a default mechanism for serializing the object.
- Since the fields are all public, they can be accessed from SQL queries (Cloudscape does not allow access to protected or private fields).
- Since the methods are all public, they can be accessed from SQL queries (Cloudscape does not allow access to protected or private methods).
- Since the class implements the compareTo() method from interface java.lang.Comparable, class Address can be indexed and ordered.
- Since the class overrides the toString() method from java.lang.Object, Cloudscape tools know how to display the value of this class's instances.

Applications would use this UDT when defining tables, for example:

```
CREATE TABLE CUSTOMER (name VARCHAR(255), addr SERIALIZE(Address));
CREATE TABLE SUPPLIER (company VARCHAR(255), addr SERIALIZE(Address));
```

This capability allows applications to define schemas with meaningful application-level column types, without resorting to a complex and nonstandard database extension language.

Java UDTs can be used in indexes, SELECT DISTINCT, GROUP BY, and ORDER BY if they implement the Java 2 compareTo() method from interface java.lang.Comparable. In this simple example, the compareTo() method performs ordering by zipcode. The query,

```
SELECT * FROM customer ORDER BY addr
```

Would return the contents of the customer table in zipcode order of addresses.

Java Stored Procedures

Cloudscape also supports Java stored procedures. A Java stored procedure consists of Java code, which is callable from SQL, runs in the database server, and accesses the database.

For the sake of example, presume that the following table exists in a database to map zip codes to cities:

```
CREATE TABLE zipmap (zipcode INT, city VARCHAR(255))
```

Then a Java stored procedure that looks up the name of a city given a zip code might look like this (error and exception handling code omitted for brevity):

```
public class ZipMap
{
    public static String cityFromZip(Connection conn, int zipcode)
    {
        PreparedStatement ps = conn.prepareStatement(
              "SELECT city FROM zipmap WHERE zipcode = ?");
        ps.setInt(1, zipcode);
        ResultSet rs = ps.executeQuery();
        if (!rs.next())
            return "Unknown City";
        String city = rs.getString(1);
        rs.close();
        ps.close();
        return city;
    }
}
```

This method would be called from SQL as follows, using the Cloudscape built-in function GETCURRENTCONNECTION to provide the connection object for the method:

```
VALUES (CLASS ZipMap).cityFromZip(GETCURRENTCONNECTION(), 94612)
```

An important thing about this example and Java stored procedures in general is that the cityFromZip() method can run on the client, the server, or the middle tier application server. There’s nothing that makes it server-specific, or even Cloudscape-specific.

Cloudscape supports the CALL statement for invoking any Java static method, including those specified as void. Cloudscape supplies JDBC OUT parameters so that values can be returned in parameters; this is done by, having the Java parameter be an array of the type desired. For example, and int parameter would be declared as int[] so that a value could be returned in it.

Using Java Methods in Constraints

Java Methods can be used in constraints. This provides a powerful way to validate data using arbitrary Java methods. For example, the ZipMap class (above) could contain a static boolean method called zipInRegion, which verifies that the zipcode is in a given geographical region (NE, SW, etc.). The following table definition could include a constraint that calls it.

```
      CREATE TABLE territory (
            region CHAR(2),
            zipcode INT
            CONSTRAINT zipValid
                  CHECK (zipcode > 0 AND zipcode < 99999)
            CONSTRAINT zipInRegion
                  CHECK ((CLASS ZipMap).zipInRegion(zipcode, region))
      )
```

In a "real" application, it is likely that the zipcode would be maintained in a CHAR(5) or a CHAR(9). In such a case it might be easier to perform the zipValid check constraint with a Java method call, too.

Java method calls in constraints can be combined with Java stored procedures. For example, the zipInRegion method could query a lookup table in the database to validate whether the region contained the zipcode.

Using Java Methods in Triggers

Similarly, Java methods can be used in triggers. Java methods in triggers can be used to provide email notification about updates, perform updates in other databases via JDBC, or literally anything that can be done in Java. Here's an example of a trigger that calls a Java method for each insert into the customer table.

```
      CREATE TRIGGER newcustomer AFTER INSERT ON customer
          CALL (CLASS Territory).addCustomer()
```

The new and old values of the modified rows are available as virtual tables. Java methods that are called as trigger actions can query the virtual tables via JDBC.

Loading Classes from the Database

Java stored procedures or other application logic can be stored in the database along with the data, and the database can be told to load the classes from there. This makes the database data and its logic into a single, self-contained application package. This simplifies application deployment, as it reduces the potential for problems with a user's class path. It is particularly useful when the classes stored in the database contain the logic for data types used in columns in the database – the database will not be dependent on anything outside of it.

Storage and loading of application classes in the database is provided by two mechanisms: the ability to add, remove and replace JAR (Java class file archive) files in the database, and the ability to add these JAR files to the class path.

Once the application code has been added to the database, moving or copying the database ensures that the appropriate application logic is moved along with the data. This means that object data will never be separated from its methods.

External Virtual Tables

An external virtual table (EVT) is a table that gets its data from some external data source rather than from the database. The results of a query on an EVT are treated exactly the same way as the results of any other query, and look exactly as if they came from a database table. This allows the integration of external data such as real-time data feeds, formatted data in operating system files, other databases (even non-SQL databases), and any other tabular data sources.

Creating an external virtual table is extremely easy. Cloudscape will treat any object that implements the java.sql.ResultSet and java.sql.ResultSetMetaData interfaces as if it were a read-only table. An example of using such a class is as follows. This EVT returns the directory contents of a zip file.

```
SELECT name, compressed_size
FROM NEW ZipEVT('c:\jdk1.1.6\lib\classes.zip') AS evt
WHERE compressed_size > 5000
ORDER BY name
```

In this example, ‘ZipEVT’ is the name of a class on the current class path which implements the result set interfaces. The “NEW ZipEVT(…)” creates a new instance of the class using a constructor that takes the provided arguments. In this case the argument passes in the name of an archive file. The “AS evt” provides a name for the virtual table.

The implementation of the class which performs the above function can be done in approximately 200 lines of Java code (it is provided as a demo in the released product).

External virtual table implementations may optionally provide cost information to the query optimizer. By default, external virtual tables are considered to be very expensive. The implementor may provide more accurate cost information by implementing the VTICosting interface. This interface provides methods that the query uses to get the estimated row count and the cost of instantiating the virtual table and iterating through its rows.

For read-write VTIs, you can insert and delete on virtual tables that implement the java.sql.PreparedStatement interface.

Since external virtual tables act just like database tables, they can be used in views. This has the benefit of adding an entry to the system catalogs so that tools can see the external virtual table as if it were a real table.

```
CREATE VIEW jdk116 AS
SELECT *
FROM NEW vti('c:\jdk1.1.6\lib\classes.zip') AS zipfile
```

Results from external queries can of course be used in all the relational operations (project, select, join, union, etc.), as well as in ordering and grouping.

Security Features

Cloudscape includes security features for deploying databases to remote sites and for integrating them into existing user authentication schemes.

Encrypted Databases

When a database is deployed to a remote or mobile location, it is not possible to use physical security to prevent unauthorized access to data. If the data files can be read, a sophisticated user could decode the information they contain. The only way to secure data in this environment is to encrypt it on disk. That way, simply being able to read the database files does not reveal the data.

Cloudscape supports secure remote data database encryption. All data in such a database is decrypted by the database engine when read and re-encrypted when written back to disk. No data exists in clear-text form in the database files. Since the entire database is encrypted, the structure of the database schema is also hidden.

For an encrypted database to be usable, a "boot password" must be provided when the database is first started. This is a separate password from the usual database connection username and password (which must also be supplied to access the database). Without the boot password, the database will not start.

Database encryption is useful for applications that distribute databases to locations where physical security of the files cannot be guaranteed. For example, mobile databases on notebook computers can be stolen if the notebook computer is stolen. Applications that are installed on remote multi-user machines are subject to unauthorized access if the remote administrator does not appropriately protect the files.

Database encryption adds less than 10% performance overhead, and takes no additional disk space. It is based on the 1.2.1 version of the Java Cryptographic Extension (JCE). DES is used as a default encryption method, or the user can configure which algorithm to use.

External User Authentication

Cloudscape also provides support for integrating into external user authentication schemes. Rather than maintaining an internal list of authorized users, Cloudscape can be configured to check with an external authentication service. LDAP and NIS+ support is provided, and custom schemes are supported via user-defined JNDI classes.

Not having the user names and passwords maintained in the database means less administrative overhead (to transfer names into the database). This is especially important in deployed server applications, which must be deployed with as little administrative overhead as possible.

LDAP (Lightweight Directory Access Protocol) is an emerging Internet standard which provides an open directory access protocol running over TCP/IP. Windows NT domain user authentication can be provided through LDAP by using the Netscape NT Synchronization Service. NIS+ (Network

Information Services Plus) is an enhanced version of the NIS name service used on the Sun Solaris operating system.

Because Cloudscape is an embedded system, it provides simple user authorization controls. Users can be restricted to read-only access or restricted from any access on a per-system or per-database level. This ensures only permitted, authenticated users access or modify a database.

Summary

This paper has described the fundamental architecture of Cloudscape, and explained how it can be embedded inside a client or a server application. It discussed the use of standard JDBC calls to manipulate the data. It showed how to store Java objects in tables and how to take advantage of Java integration to store Java objects in tables, create Java stored procedures and Java User-Defined Datatypes, and store your Java classes in the database. It summarized the security mechanisms available in Cloudscape.

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