Learning
Cloudscape:
The Tutorial

Cloudscape Version 3.0
December 15, 1999
# Table of Contents

**About This Document**
- Purpose of This Document  
- Audience  
- Tutorial Roadmap  
- Typographical Conventions

## 1 Overview

**Technology Overview**
- What Is a DBMS?  
- Relational Database Management Systems  
  - The SQL Standard  
  - Transaction Processing and More  
  - Proprietary Programming Language  
- Object-Relational DBMSs  
- The Java in Cloudscape  
  - Non-Proprietary Programming Language  
  - Run Methods Anywhere  
  - Deploy Cloudscape Anywhere  
  - Synchronization: Keeping Databases in Touch

**The JBMSTours Scenario**
- The Company  
- The Application: The *JBMSTours* Package

**Application Overview**
- The Application Queries the Data
Learning Cloudscape

2 Cloudscape Basics and the Sample Database

Getting Acquainted with Class Path

Setting the Variable vs. Using the Runtime Option

How to Specify the Location of Files in the Class Path

The CLOUDSCAPE_INSTALL Environment Variable

What to Include in the Class Path for this Tutorial

Path Separators

Task: Determine Your Class Path

Getting to Know Your System Directory

Task: Create a System Directory for the Tutorial

Task: Create a Home or Working Directory for the Tutorial

Task: Make Sure Your PATH Variable Is Correct

Task: Customize Your setclasspath Script

Running the Script

Directory Review

Getting an Application to Start Up Cloudscape

Task: Test the Class Path Script

Task: Creating, Compiling, and Running HelloWorld

Task: Experiment: Try to Get an Error

Working with Connections

Task: Get a Connection: Create a Database

Task: Shut Down Cloudscape

Task: Examine the System Directory and Information Log

Building the Database toursDB

Task: Run JBMSTours.CreateToursDB

3 Using Cloudview

Tables in a Relational Database

Working with Cloudview to Create Dictionary Objects

Task: Start Cloudview in the Home Directory
Task: Connect to HelloWorldDB 3-4
Task: Create the Sayings Table 3-4
Task: Create the Responses Table 3-5
Task: Define a Primary Key 3-6
Task: Define a Foreign Key 3-7
Task: Enter Data Graphically and with an SQL Statement 3-7
Task: Open and Execute an SQL Batch File 3-8
Task: Browse the Data in the Sayings Table 3-8

Examining Tables and Data in toursDB with Cloudview 3-9
Task: Open a Connection to toursDB 3-9
Task: Browse the Definition of the AIRLINES Table 3-9
Task: Browse the Data in the COUNTRIES Table 3-10
Task: Disconnect, Exit (and Shut Down Cloudscape) 3-11

4 Objects and SQL 4-1
Using Cloudview to Work with Stored Objects 4-2
Task: Examine the Cities Table Definition 4-2
Task: Browse the Contents of the Cities Table 4-2
Task: Browse a City Object 4-3
Executing Methods from the Object Inspector 4-3
Task: Execute an Object’s Methods 4-3
Task: View the Superclass’s Methods. 4-4
Introducing . . . SQL! 4-5
Queries 4-5
Task: Select Data from the Sayings Table 4-5
WHERE Clauses 4-6
Task: Execute an SQL-J Statement with a WHERE Clause 4-6
Joins 4-6
Task: Execute Joins 4-7
Task: Execute Methods in SQL-J Statements 4-8
Querying toursDB 4-9
Task: Execute Queries in toursDB 4-9
Task: Use a WHERE Clause 4-9
Task: Use More Complex Search Conditions 4-10
Task: Join the Countries and Cities Tables 4-10
Task: Disconnect, Exit (and Shut Down Cloudscape) 4-11

5 Cloudscape Database Applications 5-1
Cloudscape Deployment Options 5-1
What Does Embedded Mean? 5-2
Application Servers and Embedded Cloudscape 5-3
Cloudscape’s Server Frameworks 5-3
A Cloudscape Synchronization System 5-5
Working with the JDBC Interface 5-6
Applications and Queries: Statements and ResultSets 5-7
Task: Run and Examine HelloWorldApp.java 5-7
Handling SQLExceptions 5-9
Task: Change the logSeverityLevel Property 5-11
Task: Compile, Run, and Examine HelloWorldExc 5-11
Closing Statements and ResultSets 5-13
Running the BuildATour Application 5-14
Task: Run JBMSTours.BuildATour 5-14

6 Working with SQL-J’s Java Extensions 6-1
What’s the J in SQL-J? What’s ij? 6-2
Meet the City Class 6-3
Task: Examine and Execute CityTest 6-3
Work with Stored City Objects 6-5
Notes About Starting ij 6-5
Task: Start ij 6-6
Task: Connect to toursDB 6-6
Task: Execute SQL-J Statements Referencing Stored City Objects 6-6
Task: Access a Field in an SQL-J Statement 6-7
Task: Execute More Complex SQL-J Statements 6-7
Database-Side JDBC Methods 6-10
Task: Execute Methods in a Program as Application-Side Methods 6-11
Task: Execute Methods in Statements as Database-Side Methods 6-11
VALUES Expression 6-13
Task: Use the VALUES Command 6-13
Accessing Classes and Class Aliases 6-14
Task: Execute a Method in a VALUES Expression Using an Instance 6-15
Task: Execute a Method Using a Class, Not an Instance 6-16
Task: Work with Class Aliases 6-16
Task: Execute Some More Methods Using Class Aliases 6-17
Task: Use a Constructor 6-18
Static Method Aliases 6-18
Task: Create a Method Alias 6-19
Task: Execute Method Aliases for Database-Side Methods 6-19

7 Programming for Performance 7-1

Prepared Statements 7-2

Task: Prepare a Statement in ij 7-4
Task: Execute a Prepared Statement in ij 7-5
Task: Compare the Insert Performance of PreparedStatements o that of Statements 7-6

Stored PreparedStatements 7-8

Task: Create a Stored Prepared Statement 7-8
Task: Work with One of the Stored Prepared Statements in toursDB 7-9
Task: Execute a Method That Uses Stored Prepared Statements 7-11
Task: Compare the Performance of Statements, PreparedStatements, and Stored Prepared Statements 7-11

Performance, Optimization, and Indexes 7-12

Types of Queries for Which Indexes Are Useful 7-13

Task: Find the Name of the Index on the orig_airport Column 7-14
Task: Execute a Query with and without an Index 7-15
Task: Execute a Query with a More Complex Comparison 7-15
Task: Execute the Query That Uses the Column in an Expression 7-16
Covering Indexes 7-17
Task: Execute a Query in Which the Index Covers the Query 7-17
Task: Delete the Index and Re-Execute the Query 7-17
RunTimeStatistics 7-17
Task: View the RunTimeStatistics for a Query 7-18

8 Virtual Tables, External Data, and Aggregates 8-1

Virtual (Derived) Tables and Views 8-2

Task: Work with a Virtual Table 8-2
Task: Work with a View 8-3

External Virtual Tables and Bulk Import 8-5

Task: Import Data from a Flat File Using the Built-In Utility 8-6
Task: Import Data from a Flat File Using an SQL-J Statement 8-7

External Virtual Tables and External Databases 8-8

Task: Configure the ODBC Data Source for the Access Database 8-9
Task: Load the ODBC-JDBC Bridge Driver and Import Data 8-9

Aggregate Data 8-11

Task: Run JBMSTours.BuildATour a Few More Times 8-11
Task: Start ij and Work with Aggregates 8-11
Task: Work with the User-Defined Aggregate MAXBUTONE 8-13
GROUP BY and GenerateReport

Task: Use GROUP BY
Task: Use HAVING
Task: Run JBMSTours.GenerateReport

9 Working with Connections and Transactions

Transactions

Transaction Isolation and Locking

Transactions and Connections

Task: Start ij and Open a New, Named Connection
Task: Set a Database Property
Task: Open a Second Named Connection
Task: Begin a Transaction in Each Connection
Task: Commit TransactionA
Task: Repeat the Previous Tasks Using READ_COMMITTED
Task: Get Cloudscape to Automatically Roll Back a Transaction
Task: Work With the LockTable VTI

Connecting to Multiple Databases

Task: Run JBMSTours.ArchiveData

10 Storing Objects and Classes

Storing Objects in the Database

Task: Create a Class for HelloWorldDB
Task: Try Altering the Class and Storing New Instances
Task: Add the Explicit serialVersionUID to Square

The People Table and Storing Subclasses

Task: Select Instances of Subclasses from the Person Table

Storing Media Objects and Playing Them within Cloudview

Task: Play the Maps in toursDB
Task: Examine Picture.java to See How the Image Is Stored
Task: Insert Some URLPictures

Storing Classes in the Database

Task: Create a Jar File for the JBMSTours Package and Subpackages
Task: Add the Jar File to the Database as ToursLogic
Task: Alter the Value of Your Class Path
Task: Run JBMSTours.GenerateReport with Class Loading from the Database

11 Servers, Servlets, and Applets

JDBC Servers
Task: Start RmiJdbc Server
Task: Run ij as a Client Application
Task: Open a Client Window, and Run an Application

Database Servlets

About the Servlets
Appropriate Web Servers and Web Browsers
Running the Servlets
Task: Configure Cloudconnector for the Servlets and Start Cloudconnector
Task: Configure Java Web Server 1.1 for the Servlet and Start Java Web Server
Task: Use the Servlets

Task: Use the Servlets

Task: Configure and Start Cloudconnector
Task: Access the Applet from a Browser

Working with a Database Applet
Overview
About the Applet

Appendix A
Troubleshooting Common Problems
Environment Space Errors in Windows 95
I/O Errors on Solaris
ClassNotFound When Accessing Demo Database

Cloudscape Glossary
Learning Cloudscape
About This Document

- “Purpose of This Document” on page xi
- “Audience” on page xi
- “Tutorial Roadmap” on page xii
- “Typographical Conventions” on page xiv

Purpose of This Document

This book is a hands-on introduction to Cloudscape. It presents some basic Cloudscape concepts and leads you through some exercises that will help you get started using the product quickly.

Cloudscape is an object-relational database management system, which is an inherently complex product. You cannot fully learn how to use all the features of Cloudscape by performing the tasks outlined in this book. This book can, however, give you exposure to what working with the product is like.

Audience

This book is for new Cloudscape users, who are usually Java programmers. You don’t have to be a Java programmer to do this tutorial. Managers, system
administrators, and anybody who wants to learn the Cloudscape technology will also benefit from doing the hands-on tasks in this tutorial.

However, you need at least a little Java exposure to do this tutorial (see below).

**Tutorial Roadmap**

Cloudscape is a Java application designed to be used and deployed by other Java applications. This tutorial does not teach you the Java programming language or go into great detail about fundamental Java concepts such as classes and objects or inheritance. You might want to do Sun’s online Java tutorial([http://java.sun.com/docs/books/tutorial/index.html](http://java.sun.com/docs/books/tutorial/index.html)) before you do this one if you are not familiar with programming in Java.

Likewise, it is not an exhaustive study of SQL or relational databases, so you may want to consult third-party books after you finish this tutorial in order to learn more about those topics.

The lessons are designed to be done in order. However, you can do a certain amount of skipping around (you may have to go back and do one or more tasks). Every one should do lesson 2, at least, because you set up the tutorial environment in lesson 2. To do lesson 6, you must have done lesson 5 first.

Some lessons can be done as stand-alone examples.

You might want to do this tutorial online (using either the HTML or the PDF version) instead of looking at a printed version. Doing the tutorial online allows you to cut and paste queries, which you will be doing a lot.

- **Lesson 1, “Overview”**
  Concepts: Cloudscape product basics: A brief overview of Cloudscape: what it does, how you use it. Relational databases, object-relational databases, and Java. It also introduces you to the fictional JBMSTours tour company and the JBMSTours application and sample database, *toursDB*, which serve as the foundation for this tutorial.
  Tasks: None.
• **Lesson 2, “Cloudscape Basics and the Sample Database”**
  Concepts: Class path, system directory, JDBC driver, connections, shutdowns, the log.
  Tasks: Determine class path. Create working and system directories. Run an application that creates a database. Look at the information log. Create and build ToursDB.

• **Lesson 3, “Using Cloudview”**
  Concepts: Tables in a relational database, dictionary objects, SQL data types, SQL basics. Working with Cloudview.
  Tasks: View and create dictionary objects.

• **Lesson 4, “Objects and SQL”**

• **Lesson 5, “Cloudscape Database Applications”**
  Tasks: Run a basic application that uses a statement. Add SQLException checking and the shutdown command. Run BuildATour.

• **Lesson 6, “Working with SQL-J’s Java Extensions”**
  Tasks: Run CityTest, CityTest2. Execute database-side JDBC methods. Create and run method aliases.

• **Lesson 7, “Programming for Performance”**
  Concepts: The performance benefits of prepared statements and stored prepared statements and indexes.
  Tasks: Use ij’s prepared statement feature. Run a Java application comparing the performance of statements and prepared statements. Create a stored prepared statement. Analyze index benefits.

• **Lesson 8, “Virtual Tables, External Data, and Aggregates”**
  Concepts: Working with virtual tables within SQL-J statements, importing and exporting data, aggregates (set functions).
• Lesson 9, “Working with Connections and Transactions”
  Concepts: Transactions and locking. Multiple connections to a single
database and to more than one database.
  Tasks: Open two named connections in ij. Try to work with the same tables
from both connections. Test the locking schema of your product. Run
ArchiveData.
• Lesson 10, “Storing Objects and Classes”
  Concepts: Storing large data such as image and sounds. Storing objects and
working with subclasses. Storing classes in the database.
• Lesson 11, “Servers, Servlets, and Applets”
  Concepts: Servers, servlets and client applets.
• Appendix A, “Troubleshooting Common Problems”
  Quick how-tos for typical roadblocks.
• “Cloudscape Glossary”
  Cloudscape terminology.
• “Tutorial Index”
  Index to this book.

Typographical Conventions

This manual uses some typographical conventions to highlight elements of the
SQL-J language, operating system commands, the Java programming language,
and the like.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Usage</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italic</td>
<td>New terms</td>
<td>defined by keys</td>
</tr>
<tr>
<td></td>
<td>File and directory names</td>
<td>C:cloudscape</td>
</tr>
<tr>
<td></td>
<td>Java class names, objects, and</td>
<td>PreparedStatement</td>
</tr>
<tr>
<td></td>
<td>methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cloudscape dictionary objects</td>
<td>The Hotels table</td>
</tr>
<tr>
<td></td>
<td>In syntax, items that you do not</td>
<td>CREATE TABLE TableName</td>
</tr>
<tr>
<td></td>
<td>type exactly as they appear, but</td>
<td></td>
</tr>
<tr>
<td></td>
<td>replace with the appropriate name</td>
<td></td>
</tr>
<tr>
<td>Typeface</td>
<td>Usage</td>
<td>Examples</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
<td>----------------------------------------------------</td>
</tr>
<tr>
<td>bold, fixed-width</td>
<td>Syntax</td>
<td><code>CREATE TABLE tableName</code></td>
</tr>
<tr>
<td></td>
<td>SQL-J examples</td>
<td><code>SELECT hotel.getPrice()</code></td>
</tr>
<tr>
<td></td>
<td>Java application examples</td>
<td><code>ResultSet rs = ps.executeQuery()</code></td>
</tr>
<tr>
<td>roman, fixed-width</td>
<td>Comments within examples</td>
<td><code>-- This line ignored</code></td>
</tr>
<tr>
<td>ALL CAPS</td>
<td>SQL-J keywords (commands)</td>
<td>You can use a CREATE TABLE statement</td>
</tr>
</tbody>
</table>
1 Overview

Concepts covered:

- Technology overview, RDBMSs, object-relational DBMSs, Java.
- The sample application and database: *JBMS* tours and toursDB.

Tasks:

- None. Information only.

Technology Overview

Cloudscape Version 3.0 is a Java- and SQL-based object-relational database management system. Cloudscape is written entirely in Java and can be deployed as an unconnected, embedded database manager in another Java application, whether single-user or multi-user. Multi-user applications are often called application servers. Cloudscape can also be embedded inside the lightweight JDBC server framework called RmiJdbc Server, which allows multiple client JDBC applications to connect to a central database server over a network. Cloudconnector is a more powerful framework that provides JDBC, HTTP access, network security, and many other features.

Cloudsync allows periodically connected applications with their own embedded database managers and their own copies of a database to be part of a distributed system. The synchronization technology takes care of keeping the distributed copies of the databases (known as targets) in sync with the central database (called a source).

This section defines some basic database concepts such as database management system (DBMS), relational database management system (RDBMS) and object-
Overview

relational database management system (ORDBMS). It also looks at SQL, a standard query language, and the exciting features of Cloudscape’s Java extensions to SQL.

It covers the following topics:

- “What Is a DBMS?” on page 1-2
- “Relational Database Management Systems” on page 1-3
- “Object-Relational DBMSs” on page 1-4
- “The Java in Cloudscape” on page 1-5
- “Synchronization: Keeping Databases in Touch” on page 1-7

What Is a DBMS?

A database is information, and a database management system (DBMS) is the software that stores and retrieves the information in the database.

A database is not just random data; it is a collection of data pertaining to one well-defined purpose, structured in such a way that information can be retrieved from it. Imagine a company called JBMSTours. The company sells packaged and customized tours around the globe. JBMSTours constructs a database called toursDB that stores information about the different tours it offers. The database includes information about cities, hotels, and flights. Unrelated information, such as employee salaries, would be in a different database.

(Later in this tutorial, we ask you to imagine this fictional tour company more completely; see “The JBMSTours Scenario” on page 1-8.)

A database is also self-describing. A database maintains information about the data it stores and how it is organized. This information is called metadata.

The DBMS is the software that accesses, retrieves, and modifies data in a database. A separate application, often called a client application, interacts with the DBMS software to process data, present a user-friendly interface, and provide application logic.

With a Cloudscape database, as you will read in this chapter, some of the distinctions between database and application are blurred. The Java in Cloudscape means that you can store application logic in the form of Java classes in the database and run them in Cloudscape if appropriate, or in the client application, or both.

But more about that later.
Relational Database Management Systems

Cloudscape is an object-relational database management system. A relational database is a database that presents information in one or more tables containing data about a particular entity (such as hotels, cities, or countries). Separate tables in the database are related to each other, so the data in the database can be accessed (or queried) from a single table or from multiple tables at the same time.

The SQL Standard

Cloudscape supports entry-level SQL-92. SQL is the standard query language used with relational databases. (When it was originally created, SQL stood for structured query language, but it does not stand for anything nowadays.) No matter how a particular RDBMS has been implemented, the user can design databases and insert, modify, and retrieve data using the standard SQL statements and well-defined data types. SQL-92 is the version of SQL standardized by ANSI and ISO in 1992. Entry-level SQL-92 is a subset of full SQL-92 specified by ANSI and ISO that is supported by nearly all major DBMSs today.

Transaction Processing and More

An RDBMS provides transaction control, a set of rules and guidelines that ensure that integrity is maintained during and after operations on data. Transaction control is what protects your bank account from being $200 short when you cancel an ATM withdrawal.

Cloudscape, like RDBMSs, provides transaction control, plus automatic crash recovery and runtime rollback.

You can use Cloudscape as an RDBMS, because it supports the basic RDBMS data types, supports the SQL standard, allows you to store data in related tables, and supplies transaction management. Some users simply want a standard relational database that runs in a JVM. For them, Cloudscape is the easy choice.

Proprietary Programming Language

RDBMSs did not originally provide any programmability. They provided a way to store and retrieve data and the ability to run a limited set of functions on the data. Customers implemented logic at the application level—not in the database, as shown in Figure 1-1.
Originally, there were two good reasons for separating these tasks. First, it let each program do the task it was best suited for. Second, it allowed several users with different goals—and perhaps different applications and application logic—to access a central repository of data.

RDBMS vendors later provided some limited programmability, typically with a proprietary language based on SQL that allowed users to implement a simple logic within the database itself. Stored procedures and triggers are both examples of server-side logic.

Cloudscape does provide programmability in the DBMS, but its capabilities are much more powerful than those of an RDBMS. You will read more about this in the next section.

**Object-Relational DBMSs**

SQL-92 restricts stored data to the integer, floating-point number, character string, date-time, numeric, and decimal data types. Each basic data type provides a fixed number of functions and operators.

Cloudscape is an object-relational DBMS, or ORDBMS. Object-relational DBMSs provide object extensions to the traditional RDBMS functionality. An object-relational database can store and process user-defined objects, which are modules from object-oriented programming, an evolutionary form of modular programming that allows pieces of software to be reused and interchanged between programs. An object consists of both data and corresponding behavior. In object-oriented programming, you do not separate data from the logic that goes with it.

Some RDBMSs store complex objects as “blobs” (binary large objects), which means they cannot execute any of the object’s behavior, called methods. A blob
isn’t really an object at all. Searching for blobs without specialized ways to refine the search makes it difficult to retrieve an object from a database.

In an ORDBMS, however, you can use an object’s behavior to assist in object retrieval. A very simple example of a “complex” data type is the City data type from our toursDB sample database, which you will be building later in this tutorial. The City definition includes several methods. For example, you could query the database for a city in the Southern Hemisphere where the average temperature in July is above 80 degrees Fahrenheit.

The Java in Cloudscape

The beauty of Cloudscape is its integration with Java. The complex objects stored in Cloudscape and the stored procedures are written in the Java programming language.

Java is an extremely popular and portable programming language. The Java in Cloudscape makes it powerful in several ways:

• It is a non-proprietary programming language.
• It is the same language that runs in client applications, so you can run methods in the database or in the application.
• Because Java is practically a platform, you can deploy Cloudscape just about anywhere.

Java is an object-oriented programming language developed by Sun Microsystems. Applications written in Java do not interact with a computer’s operating system directly. Instead, byte code from compiled Java applications is interpreted by the Java Virtual Machine (JVM). Java applications can run on any computer that has a JVM. In the future, Java applications may be able to run on computers or small devices that contain a “Java chip.”

A basic concept in Java is the class. A class is a template for an object; an object is an instance of a class. You create an object by instantiating a class. For example, JBMSTours defines a template for city objects in the City class. The class definition encapsulates all the variables and operations of the City class. In Java, operations are called methods. A particular city, such as Santiago or Paris, is a particular instance of a City. You can create an instance of the City class for these cities; these instances are called objects.

Object-oriented languages are popular because they allow programmers to re-use code. It is easy to re-use or use somebody else’s class definitions because Java classes include published interfaces. That is, in order to use a class, you do not have
Overview

to know internal details. All you need to know is how to use the methods that allow an object of that class to interact with other objects. Even if a programmer changes the internal details, the interface stays the same.

Most Java programmers take advantage of rich collections of existing classes and methods in Java class libraries. Class libraries are provided primarily by compiler vendors, but many class libraries are supplied by independent software vendors. Some class libraries are considered essential to the Java platform: the Java base and core classes, which are included with a Java Development Kit (JDK).

One of the core classes that are important in database applications is the java.sql package, the set of classes that form the JDBC API. JDBC is a Java API (application programming interface) for executing SQL statements. You use this standard API to work with Cloudscape databases.

The bibliography in the preface of this book provides a list of good sources for learning about the Java programming language.

Non-Proprietary Programming Language

You do not have to “translate” a Java object into a proprietary format to store it in a Cloudscape database. You can store Java data types in a Cloudscape database. Because of Java’s popularity and reusability, not only can you write your own data types, you can also easily purchase third-party data types that are immediately usable in a Cloudscape database.

Run Methods Anywhere

Because Cloudscape is itself written in Java, it runs within a JVM in the same way as any other Java application does. This makes it easy to migrate user objects from the application to the DBMS and vice versa. A method can run in the application or in Cloudscape. All that is required is to make the application’s class files available to the JVM in which Cloudscape is running.

Do you remember Figure 1-1 on page 1-4, which showed application logic as separate from the data? With Cloudscape, the picture looks different, as in Figure 1-2 on page 1-7.
Java objects can run in the client application or in Cloudscape and can be stored in the database.

You do not have to declare these methods within the Cloudscape system, although you can provide aliases for them that makes it easier to run them.

**Deploy Cloudscape Anywhere**

Cloudscape is a lightweight database with a small footprint. This means that it doesn’t use very much memory or disk space, and it is easy to administer.

Normally you wouldn’t care what language a DBMS was written in. But Java is more than just a programming language; in some senses, Java is a platform. Because Cloudscape is written in Java, Cloudscape can run in any environment that supports a JVM, or Java Virtual Machine. Cloudscape databases are portable across platforms and easy to manage.

Cloudscape’s deployability means that you can embed it in any Java applications. Applications do not need to connect to a central database; they can carry one with them. When Cloudscape is embedded in an application, the application makes local calls to the DBMS. No network connection or networking software is needed.

You can also deploy Cloudscape and Cloudscape applications in a client/server environment. For more information about different Cloudscape deployment possibilities, see “Cloudscape Deployment Options” on page 5-1.

**Synchronization: Keeping Databases in Touch**

Cloudscape synchronization allows you to “tear off” partial or complete copies of databases and keep them connected to a corporate system. Any Cloudscape database can be part of a distributed Cloudscape system.

Cloudscape synchronization is a process in which Cloudscape copies dictionary objects, data, and Java objects from a source database to and from one or more
Overview

target databases and controls consistency of data. Because Cloudscape synchronization is message-based, remote databases do not have to be continuously connected over a network. Remote target databases need only connect intermittently, long enough to send and receive HTTP messages that refresh local data with up-to-date data from a central database.

The advantages of a distributed system are:

- Better performance, especially response time. Since the data is accessed locally, there is no network latency, the DBMS can cache data that the application is actually using, and the local CPU, memory, and disk resources can be fully utilized without conflict with other users.
- Nomadic operation. Since the data lives on the remote computer, the computer does not need to be continuously connected to the network. For example, if the computer is a laptop, the application will run the same way whether it’s connected to the HQ network or out in the field.
- Improved security. Since the application does not have to connect to a database server, there is no need to make “holes” in the corporate firewall for database network access. Since different data can be sent to different users, it can be arranged that no user could possibly see data not meant for him or her.

For more information about Cloudscape synchronization, see the Cloudscape Synchronization Guide.

The JBMSTours Scenario

Throughout this tutorial, we will be asking you to work with a Java application and database automatically installed with Cloudscape. The sample database is called toursDB; the sample application consists of a package of Java classes called JBMSTours. You will learn more about the specifics of this application in “The Application: The JBMSTours Package” on page 1-10.

To make the most sense of the example, imagine the following scenario:
Our fictional company, JBMSTours, is a travel company specializing in do-it-yourself customized tours. The company distributes software that allows customers to book complete tours to two cities of their choice, including hotel stays as well as flights.

JBMSTours has chosen to write the software in Java, because Java frees the firm from a specific operating system or type of hardware; the software can run on any computer with a JVM. Since the application is data-centric (it must find information about cities, and about available flights, and hotels), it needs to access a database. The company doesn’t want customers to have to be connected to a central database server or wait for slow connections over the Internet; instead, JBMSTours wants each copy of the application to have its own copy of the database and the necessary database management software.

JBMSTours has chosen Cloudscape as its database management software, for the following reasons:

- Cloudscape is written entirely in Java and can be “embedded” in the distributed software as simply another set of Java class files, invisible to the user.
- Cloudscape does not require the special installation or configuration of other personal database software, making the entire package—the JBMSTours software and the underlying database software—easy to use.
- Cloudscape’s object-relational technology and Java extensions mean that where appropriate, JBMSTours can store data exactly as the application uses it—as Java objects. It doesn’t have to “translate” business models into other kinds of data.
- Cloudscape synchronization means that clients can keep databases up to date and upload any changes made locally. Synchronization is done in the background without any input needed from the user. (The current version of this tutorial does not address synchronization.)
The Application: The JBMSTours Package

The fictional company’s fictional application and database are included with your installation. The application consists of a set of class files, with a few main or entry classes (which contain `main()` methods). Javadoc for the application is also included; we encourage you to examine the javadoc as you do this tutorial. Javadoc is automatically generated HTML documentation about a Java class or package.

You may also look at the Java source files, which are also provided.

- the javadoc
  is located in the `demo/programs/tours/javadoc` directory in the cloudscape base directory
- the Java source files
  are located in the `demo/programs/tours/JBMSTours` directory in the cloudscape base directory
- the compiled classes
  are located in the `demo/programs/tours/JBMSTours` directory in the cloudscape base directory

Application Overview

Taking advantage of the object paradigm, the JBMSTours application builds a `Tour` object for the customer using it to create and book a tour. For the sake of simplicity, a `Tour` object in the example application is a two-city tour that includes a flight from the user’s home city to the first city, a flight to the second city, and a return flight back home. It also includes a hotel stay in each of the two cities visited.

The classical reason for using a DBMS is that it provides the following:

- querying
  It allows you to query complex and large data stores for specific information.
- persistence
  It allows you to make your data persistent by storing it.

The sample application uses Cloudscape for both purposes. It queries the databases to get information about cities, flights, and hotels, and it stores information in the database to make it persistent.
The Application Queries the Data

When generating a Tour object, the application queries the database for information about countries, cities, hotels, flights, and airlines.

JBMS Tours stores a repository of core data in the toursDB database:

- **countries**
  Stored as SQL-92 data types in the Countries table. The application only reads this data; it does not change or add to this data.

- **cities**
  Stored as Java objects in the Cities table. The application only reads this data; it does not change or add to this data.

- **hotels**
  Although these are Java objects in the application, they are “translated” into flat data in the Hotels table. (Translating an object into flat data is sometimes called object-relational mapping.) Information about a hotel’s availability on specific dates is stored as SQL-92 data types in the HotelAvailability table. The application only reads the Hotels table, but as it books tours it writes to the HotelAvailability table.

- **flights**
  Stored as SQL-92 data types in the Flights table. Information about a flight’s availability on specific dates is stored as SQL-92 data types in the FlightAvailability table. The application only reads the Flights table, but as it books tours it writes to the FlightAvailability table.

- **airlines**
  Stored as SQL-92 data types in the Airlines table. The application only reads the Airlines table.

The application queries these tables to construct a Tour object.

A Tour object itself is made up of other Java objects—CustomerFlights, HotelStays, dates, and some other data.

The Application Stores Tour Data

After a user constructs a tour of his or her choice, the application writes some information to the database.

- **tours**
  Once the entire tour is booked, the application stores the Tour object itself in a table called CustomizedTours.
Overview

- **hotel bookings**
  The application represents customer- and date-specific information about a hotel stay as a `HotelStay` object, which is stored only as part of a `Tour`. As it books a tour, the application stores some information about a `HotelStay` in the `HotelBookings` table.

- **flight bookings**
  The application represents customer- and date-specific information about a flight as a `CustomerFlight` object, which is stored only as part of a `Tour`. As it books a tour, the application stores some information about a `CustomerFlight` in the `FlightBookings` table.

- **groups**
  The application represents the group of people booking a tour as a `Group` object. The application translates `Groups` into flat data and store them in the table called `Groups`.

- **people**
  The application represents the individual people in a group as `Person` objects (or as an `Adult` or `Child` objects, which are subclasses of `Person`). The application stores these objects in the `People` table.
  People are mapped to Groups in the `Groups_People` table.

The Application May Use Synchronization

In a Cloudscape synchronization system, information stored in tables by an application would be available at the company office and subsequently to the airlines and hotels with which JBMSTours does business. Depending on how the back office processes the data, replicating data allows the application to actually book tours. The application does not yet take advantage of synchronization.

One “Temporary” Table: FlightObjects

The `JBMSTours` application treats flights as Java objects. However, it stores basic flight information as SQL-92 data types, because of the large quantity of flight data. In the current release, it is faster to search tables holding SQL-92 data types because you can index the data. An index is a database object that allows Cloudscape to find specific data more quickly. You cannot currently index Java data types or Java methods.

Before it builds a tour, the application searches the large `Flights` tables and generates several `Flight` objects based on the airport codes of cities the customer will visit. The appropriate airport columns are indexed in the table.
However, when actually constructing a tour, the application wants to select the appropriate flight (or a set of flights that make up a transfer flight) based on more complex criteria. The application represents a flight as a *Flight*, *OneStopFlight*, or *TransferFlight* object. It stores them in a table called *FlightObjects*. When the customer actually chooses a flight, the application can search through the small *FlightObjects* table and use the *Flight* object’s rich set of methods to choose the best flight for the customer.

For example, for a customer visiting Paris, the application cleans out the *FlightObjects* table. Then it searches for flights from the airport of the customer’s home city and to the airport in Paris and constructs a set of *Flight* objects. These various flight objects might be direct flights, one-stop flights, or transfer flights. When it comes time to build the tour, the application can search the *FlightObjects* table based on methods such as *getTotalDuration* or *getRate* to choose the best flight for the customer.

**What You Will Be Doing with JBMSTours**

In the lessons that follow, you will:

- **build the sample database**
  by running `JBMSTours.CreateToursDB` in Lesson 2, “Cloudscape Basics and the Sample Database”
  
  (The installation includes a pre-built version of *toursDB*. However, for pedagogical reasons, this tutorial asks you to build the database anew.)

- **examine the sample database**
  with Cloudview in Lesson 3, “Using Cloudview”

- **query the sample database**

- **run and examine the customer end-application**
  called `BuildATour`, in Lesson 5, “Cloudscape Database Applications”

- **execute some JBMSTours methods within SQL statements**
  in Lesson 6, “Working with SQL-J’s Java Extensions”

- **learn about improving performance**
  in Lesson 7, “Programming for Performance”

- **create views, import and export data, and generate a report**
  in Lesson 8, “Virtual Tables, External Data, and Aggregates”
Overview

- work with multiple connections
  in Lesson 9, “Working with Connections and Transactions”
- use Java’s serialization compatibility mechanisms, retrieve stored images, and store application logic in the database
  in Lesson 10, “Storing Objects and Classes”
- try out servlets or applets
  in Lesson 11, “Servers, Servlets, and Applets”

Or Look on Your Own . . .

You can bypass the lessons in this tutorial. If you just want to build the sample database and run the end-user applications, run them in this order:

- JBMSTours.CreateToursDB
- JBMSTours.CreateScript
- JBMSTours.BuildATour
- JBMSTours.GenerateReport
- JBMSTours_ARCHIVE
- JBMSTours.RUNTIME
- JBMSTours.ADMINHELPER

In a Client/Server Environment

In this tutorial, you will be working in an embedded environment. That is, you will work with an application that starts up an instance of Cloudscape. However, all the JBMSTours programs can be run as client applications with either RmiJdbc Server or Cloudconnector (purchased separately).

To run these applications as client applications to a running server, provide two command-line arguments when you run them:

- hostname
  the hostname where your server framework is running
- port number
  the port number on which your server framework is listening

If you are using the RmiJdbc framework, add the single character r.

For example:
This tutorial does not teach you how to start a server. The *Cloudscape Server and Administration Guide* can help you get started with that task.
2 Cloudscape Basics and the Sample Database

Concepts covered:

- **Class path overview.**
  “Getting Acquainted with Class Path” on page 2-2. If you already know all about class path, skip most of this section, but do determine your class path (see “Determine Your Class Path” on page 2-5).

- **System directory overview.**
  “Getting to Know Your System Directory” on page 2-6.

- **How does a Cloudscape application interact with Cloudscape?**
  “Getting an Application to Start Up Cloudscape” on page 2-10.

- **Connections and Shut Downs. The information log.**
  “Working with Connections” on page 2-13.

- **Introduction to the sample database**
  “Building the Database toursDB” on page 2-17.

Tasks:

- “Determine Your Class Path” on page 2-5.
- “Create a System Directory for the Tutorial” on page 2-7.
- “Create a Home or Working Directory for the Tutorial” on page 2-7.
- “Creating, Compiling, and Running HelloWorld” on page 2-11.
- “Experiment: Try to Get an Error” on page 2-13.
- “Get a Connection: Create a Database” on page 2-14.
- “Shut Down Cloudscape” on page 2-15.
- “Examine the System Directory and Information Log” on page 2-17.
- “Run JBMSTours.CreateToursDB” on page 2-18.
Getting Acquainted with Class Path

When you request that a JVM (Java Virtual Machine) execute a Java program, it searches through the class path, a list of directories and zip files that tell the JVM where to look for the program. If your program (or any classes used by the program) is not in the class path, the program does not work. You will get a message like the following:

Can’t find class YourProgram

This section covers the following topics:

- “Setting the Variable vs. Using the Runtime Option” on page 2-2.
- “How to Specify the Location of Files in the Class Path” on page 2-2.
- “What to Include in the Class Path for this Tutorial” on page 2-4.
- “Path Separators” on page 2-5.

And the following task:

- “Determine Your Class Path” on page 2-5.

Setting the Variable vs. Using the Runtime Option

You will need to set your class path in order to run CreateToursDB, the first application in the JBMSTours package.

There are two ways to set the class path:

- Set the operating system’s CLASSPATH environment variable, either permanently or temporarily.
- Set the class path with the runtime option (specifying the class path at the time you start your Java application and the JVM).

In this tutorial, you will set the environment variable temporarily with a script. You will need to run this script every time you open a new command window.

How to Specify the Location of Files in the Class Path

The JVM (compiler or interpreter) needs to know the path (operating system–specific instructions about location) of every class file needed by your application. Class files can live in:
• **Directories**
Class files can live in directories that indicate the package name. For example, the class files in the package **JBMSTours** live in a directory called **JBMSTours**. Class files in the package **COM.cloudscape.types** would live in the directory **types**, which would live in the directory **cloudscape**, which would live in the directory **COM**.

When indicating the path to class files that live in a directory, do not include the name of any of the directories that constitute the package name. Instead, include the directory one level above the package name directory. For example, if the directory **JBMSTours** lived on your C drive in a directory called **tours**, you would add “c:\Tours” to the class path.

• **zip or jar files**
Class files can also live in zip or jar files. Class files in zip or jar files live inside directories that have the same name as the package name within the zip or jar files. The zip or jar files can live in any directory.

When indicating the path to a class file that lives in a zip or jar file, include the path to the file, including the file name itself. For example, for a jar file called **cloudscape.jar** that lives in the **cloudscape\lib** directory of your C drive, you would add “c:\cloudscape\lib\cloudscape.jar” to the class path.

• **For JDK 1.1, JDK class files**
All Java programs need access to JDK class files. For JDKs prior to 1.2, you sometimes need to include the JDK class files in your class path. Pre–JDK 1.2 class files are located in **classes.zip** in the **/lib** subdirectory of the directory where you installed the JDK. For example:

```
c:\jdk1.1.5\lib\classes.zip
```

The pre–JDK1.2 JVM always includes this zip file in the class path unless you use the runtime override option. We won’t use that method, so adding this zip file is optional.

The JDK1.2 JVM uses a different library (**rt.jar**), and it always includes this library in the class path. Do not add it to the class path.

**NOTE:** If you operating system supports it, it is a good idea to use JDK 1.2.x for this tutorial.

### The CLOUDSCAPE_INSTALL Environment Variable

Some versions of the installer automatically create an environment variable called **CLOUDSCAPE_INSTALL** and set its value to the cloudscape base directory, the
directory in which you installed the product. If your version of the installer did not set this environment variable, do it now.

For example, if you installed the product in \cloudscape, set CLOUDSCAPE_INSTALL to \cloudscape.

**What to Include in the Class Path for this Tutorial**

This section describes what you need to include in the class path in order to do this tutorial correctly.

- **Current directory**
  Indicated with a period (.)

- **The Cloudscape jar files**
  In this tutorial you will be working with Cloudscape applications, which use the Cloudscape classes. These class files are located in jar files installed in the /lib subdirectory of CLOUDSCAPE_INSTALL.
  Include cloudscape.jar and tools.jar.
  For example:
  ```
  %CLOUDSCAPE_INSTALL%\lib\cloudscape.jar
  %CLOUDSCAPE_INSTALL%\lib\tools.jar
  ```

**NOTE:** If you are using a pre-1.2 JDK, you also need to download Cloudscape’s swingall.jar library and add it to your class path. This library is available for download from [http://www.cloudscape.com/support/Downloads/](http://www.cloudscape.com/support/Downloads/). You need to use JDK 1.1.6 or higher.

- **Your application (JBMSTours)**
  In this tutorial, your application is the JBMSTours application, and you must include the path to the application in your class path. The JBMSTours application resides in the demo/programs/tours subdirectory of CLOUDSCAPE_INSTALL. For example:
  ```
  %CLOUDSCAPE_INSTALL%\demo\programs\tours
  ```

- **license.jar, if you are using the evaluation copy of Cloudscape**
  For example:
  ```
  %CLOUDSCAPE_INSTALL%\lib\license.jar
  ```
Getting Acquainted with Class Path

Figure 2-1 Setting your class path. (JDK library not shown.) In order for Cloudscape applications to work, your class path must include the Cloudscape libraries. The sample application requires the JBMSTours library, which is in /demo/programs/tours.

Path Separators

You specify the list of paths as a single string. You must use a separator between paths; the character used as the separator depends on your operating system.

- **DOS and Windows platforms:**
  - ; (semicolon)
- **UNIX platforms:**
  - : (colon)

Determine Your Class Path

Take the time now to determine the exact text of your class path. To review, it should include:

- path to JDK (optional)
- path to primary Cloudscape library and /lib/tools.jar
- path to demo/programs/tours
- current directory
- license.jar, if you are using the evaluation version of Cloudscape

Write the text of your class path in the space provided below.
Cloudscape Basics and the Sample Database

The following is an example class path on a Windows system, where the cloudscape base directory is \cloudscape:\

.;c:\JDK\lib\classes.zip;%CLOUDSCAPE_INSTALL%\lib\cloudscape.jar;%CLOUDSCAPE_INSTALL%\lib\tools.jar;%CLOUDSCAPE_INSTALL%\demo\programs\tours;%classpath%

The following is an example class path on a Windows system, where the cloudscape base directory is C:\cloudscape and the user is working with the evaluation version of Cloudscape:

.;c:\JDK\lib\classes.zip;
%CLOUDSCAPE_INSTALL%\lib\cloudscape.jar;
%CLOUDSCAPE_INSTALL%\lib\tools.jar;
%CLOUDSCAPE_INSTALL%\demo\programs\tours;
%CLOUDSCAPE_INSTALL%\lib\license.jar;%classpath%

The following is an example class path on a UNIX system:

.:/JDK/classes.zip:$CLOUDSCAPE_INSTALL/lib/
cloudscape.jar:$CLOUDSCAPE_INSTALL/lib/
tools.jar:$CLOUDSCAPE_INSTALL/demo/programs/tours:$CLASSPATH

[space for your class path]

The rest of this tutorial will refer to your class path as yourClassPath. You will be setting that in a script (see “Customize Your setclasspath Script” on page 2-8).

Getting to Know Your System Directory

When an application creates a database, Cloudscape puts the database (a directory that holds several files) in what is known as the system directory.

The system directory is by default the current directory—whatever directory you happen to be working in. This means that when you run an application that creates a database, that database will be created in the current directory. And when you run an application that interacts with an existing database, it expects the database to be in the current directory.
The system directory also holds the information log (*cloudscape.LOG*), which is generated whenever an application interacts with a Cloudscape database.

It is a good idea to specify the system directory instead of letting Cloudscape assume the default directory. That way, your applications will work no matter where you run them.

In this section, you will create a system directory so that you can specify the system directory later. Your databases will be built in this directory. You will also create a working directory. This is the directory from which you will run scripts and Java programs. This section includes the following tasks:

- “Create a System Directory for the Tutorial” on page 2-7
- “Create a Home or Working Directory for the Tutorial” on page 2-7

**Create a System Directory for the Tutorial**

Create a system directory for the tutorial anywhere on your hard disk. Call it *tutorial_system*. This directory will contain the databases that you create in this tutorial as well as the information log and the properties file.

You will refer to the system directory on the command line when you run *CreateToursDB* later. You will need to specify the full path to the directory. For example, if you created the directory on C:\, you will refer to the system directory as:

```
c:\tutorial_system
```

The remainder of this tutorial refers to the complete path to your system directory as *your_tutorial_system*.

**Create a Home or Working Directory for the Tutorial**

1. Create a home or working directory for the tutorial activities now anywhere on your hard disk. Call it *tutorial_home*.

The home directory will be your home base for doing command-line activities in this tutorial. In other words, unless otherwise directed, issue all commands from this directory.

The remainder of this tutorial refers to the complete path to your working directory as *your_tutorial_home*.
2 /demo/programs/tours contains a subdirectory called scripts. Copy all the files in scripts (not the directory itself) to your working directory. These files should be directly inside the working directory. Having copies of these files in the home directory will make it easier to work with these files and will allow you to leave a “clean” copy in the original directory in case you want to start over later.

**Make Sure Your PATH Variable Is Correct**

The PATH variable enables your operating system to find the appropriate executables from any directory. You must have the path to the JDK that you wish to use in the PATH variable. See the installation instructions that came with your JDK for information on how to set PATH.

**Customize Your setclasspath Script**

The files that you copied over from the scripts directory include three sample script files with the command to set the CLASSPATH environment variable. The file name and command depend on your operating system.

1. Open the appropriate file from the your_tutorial_home directory.
   - Windows users should open setclasspath.bat.
   - UNIX users should open setclasspath.sh.
   - UNIX users who are working in cshell should open setclasspath.csh

2. Alter the script to set CLASSPATH to what you determined it should be in “Determine Your Class Path” on page 2-5. For example, you may need to change the path of the cloudscape base directory.

**Running the Script**

Later in this tutorial, you will be asked to run this script. Here’s how you do that:

- **On Windows:**
  
  ```bash
  setclasspath
  ```

  On Windows 95, if you get an “out of environment space” error, you need to enlarge the environment space. See Appendix A, “Troubleshooting Common Problems”, for more information.
• On Sun Solaris csh:
  source setclasspath.csh
• On sh or ksh:
  . setclasspath.sh

Directory Review

By the time you get here, you should have created two directories:

• your home or working directory (your_tutorial_home)
  This is the directory from which you will run scripts and Java programs.
• your system directory, (your_tutorial_system)
  This directory will contain two databases that you will create in this lesson.

Figure 2-2 shows the basic layout of the two directories plus the cloudscape base directory.
Getting an Application to Start Up Cloudscape

A Cloudscape application is a Java application that interacts with Cloudscape using the standard JDBC API (application programming interface). The JDBC API is part of the JDK and is not provided by Cloudscape. It consists of the `java.sql` package, which is a set of classes and interfaces that make it possible to access databases from a Java application.

In order to interact with Cloudscape, an application must first start up the Cloudscape JDBC driver. A JDBC driver is an application that implements the JDBC interface and is loaded by the JDBC driver manager. The name of embedded Cloudscape’s JDBC driver is `COM.cloudscape.core.JDBCDriver`. 

**Figure 2-2** The layout of three directories: your working directory (home to your scripts and Java programs), your system directory (home to your databases, information log, and properties files), and the cloudscape base directory (which is the value of the CLOUDSCAPE_INSTALL environment variable).
The method you use to start up the driver is not actually part of the `java.sql` package; it is the only method for interacting with Cloudscape that is not part of that package. To start up Cloudscape and make it available for connections, you pass the name of the JDBC driver to the `Class.forName` method, which you will do in the following example.

**Test the Class Path Script**

Cloudscape provides a utility to test whether your class path is correct for a particular environment. In this tutorial, you work mostly in an embedded environment. Therefore you can run the utility with the following arguments:

- embedded
- sampleApp (to test for the sample application classes)

1. Open a command window and change directories to `your_tutorial_home`.
2. Run your `setclasspath` script (see “Running the Script” on page 2-8).
3. Run the utility:

   ```
   java COM.cloudscape.tools.sysinfo -cp embedded sampleApp
   ```

   If all the needed libraries are present in the class path, the utility displays a message indicating success. If any of the needed libraries are missing, it displays a message indicating what is missing.

**Creating, Compiling, and Running HelloWorld**

1. Open the `.java` file called `HelloWorld.java` in `your_tutorial_home` directory (which you should have copied over from `scripts`). This file is the `HelloWorld` application, which you will compile and run. Figure 2-3 shows the file.
Figure 2-3  The *HelloWorld* application

The *HelloWorld* application attempts to load the Cloudscape JDBC driver. If it succeeds, it prints a message. If it fails, it prints an error message.

2 Open a command window and change directories to *your_tutorial_home*.

3 Run your *setclasspath* script (see “Running the Script” on page 2-8).

4 Compile the program. The following program uses the JVM included with the standard JDK:

   ```java
   javac HelloWorld.java
   ```

5 Run the program, using the -D parameter to set *cloudscape.system.home*:

   ```java
   java -Dcloudscape.system.home=your_tutorial_system HelloWorld
   ```

   For example, if your tutorial system directory is d:\mysystem, you would invoke the program like this:

   ```java
   java -Dcloudscape.system.home=d:\mysystem HelloWorld
   ```

   Java class names are case sensitive.

If the program successfully loads the Cloudscape JDBC driver, it prints the following message:

   *Loaded the cloudscape JDBC driver. Hello, World!*

If the program prints out the error message instead, check your class path. Reread “Getting Acquainted with Class Path” on page 2-2 if you got your class path wrong.

6 Keep the command window open for the rest of this lesson.
Experiment: Try to Get an Error

As an experiment, deliberately try to get an error with this program. Here are some possibilities:

- Deliberately omit the main Cloudscape engine zip file from yourClassPath.
  The JVM will not be able to find Cloudscape.
  Be sure to restore it after trying this!
- Change slightly the text of the Cloudscape JDBC driver name (COM.cloudscape.core.JDBCDriver) in HelloWorld.java, then recompile the program.
  The JVM will not be able to find the Cloudscape driver class.
- Move the main Cloudscape engine zip file from its current location.
  The JVM will not be able to find Cloudscape because its location is not in the class path.
  Be sure to move it back after trying this!

All three possibilities will cause the program to fail because the JVM will not be able to find the Cloudscape JDBC driver.

Working with Connections

Once an application has loaded the Cloudscape JDBC driver, the application can connect to an existing database or create a new database. All of the application’s interactions are then handled through the JDBC API, the classes and methods of java.sql.

Applications connect to databases (and create them, if they do not exist) with the java.sql.DriverManager.getConnection method. Applications specify a database connection URL as a String parameter to this method. Calling this method with the appropriate URL string returns a connection to a database.

Cloudscape’s database connection URL consists of four parts, as shown in Figure 2-4.
Cloudscape Basics and the Sample Database

Figure 2-4  Syntax of the Cloudscape database connection URL

The first two never change; the third part (which is optional) indicates the name of the database you want to connect to. The fourth part consists of zero or more attributes. An attribute is how we will specify that we want to create a new database (instead of connecting to an existing one).

In this section, we will create a database called *HelloWorldDB*. Creating this database requires a database connection URL as shown in Figure 2-5.

Get a Connection: Create a Database

In this project, you will run a Java program that creates a new database.

1. Open *CreateWorldDB.java*, which includes the new code shown in Figure
Working with Connections

2-6.

```java
import java.sql.*;

public class HelloWorld {
    public static void main(String[] args) {
        try {
            Class.forName("COM.cloudscape.core.JDBCDriver");
            System.out.println("Loaded the cloudscape JDBC driver. Hello, World!");
            Connection conn = DriverManager.getConnection("jdbc:cloudscape:HelloWorldDB;create=true");
            System.out.println("Created and connected to database HelloWorldDB");
        } catch (Throwable e) {
            System.out.println("Exception thrown.");
            System.out.println(e);
        }
    }
}
```

This is the new code that creates the database and gives you a connection to it.

**Figure 2-6** This Cloudscape database connection URL creates a new database called *HelloWorldDB*.

2 Return to your open command window and compile the file:

`javac CreateWorldDB.java`

3 Run the program, using the `-D` parameter to set `cloudscape.system.home`:

`java -Dcloudscape.system.home=your_tutorial_system CreateWorldDB`

**Quiz:** What would happen if you forgot to set `cloudscape.system.home`?

**Answer:** Cloudscape would build the database in the current directory.

4 If it runs successfully, you should see the following output:

```
Loaded the cloudscape JDBC driver. Hello, World!
Created and connected to database HelloWorldDB
```

### Shut Down Cloudscape

As you saw, starting up Cloudscape was a separate step from getting a connection to a database. Shutting down Cloudscape is likewise a separate step.
Cloudscape Basics and the Sample Database

It is important to shut down Cloudscape when the application is through interacting with embedded Cloudscape, because the shutdown command ensures that all changes are written from the database log (a temporary holding place for transactions) to the actual database. If you do not shut down Cloudscape, the next time it starts up, it takes the time to run recovery on the databases in the system. Recovery means restoring a database to its last committed state after a system failure (or stopping the application without a shutdown command) based on information stored in a special area called the database log.

In this task, you will delete HelloWorldDB, edit the CreateWorldDB program to include disconnect and shutdown commands, and rerun the program.

1. Delete the HelloWorldDB directory in the your_tutorial_system directory. (Use operating system commands).
2. Edit the CreateWorldDB.java file by adding the following code:

   ```java
   conn.close();
   DriverManager.getConnection("jdbc:cloudscape;;shutdown=true");
   ```

   **NOTE:** These two lines go at the end of the try block, right before the curly brace, and right after the following line (line 13):

   ```java
   System.out.println(
       "Created and connected to database HelloWorldDB");
   ```

3. Return to your open command window and recompile.
4. Run the program:

   ```bash
   java -Dcloudscape.system.home=your_tutorial_system CreateWorldDB
   ```

   This time, if the program runs successfully, it will display an error message as follows:

   ```java
   Loaded the cloudscape JDBC driver. Hello, World!
   Created and Connected to the HelloWorldDB database
   exception thrown
   SQL Exception: CloudscapeSystemShutdown
   at COM.cloudscape.$18.$893.$11187(Unknown Source)
   at COM.cloudscape.$18.$893.$11187(Unknown Source)
   at COM.cloudscape.$18.$893.$1153(Unknown Source)
   at COM.cloudscape.$18.$1068.connect(Unknown Source)
   at java.sql.DriverManager.getConnection(DriverManager.java:91)
   at java.sql.DriverManager.getConnection(DriverManager.java:149)
   ```
Building the Database toursDB

The output shows an error because successful shutdown commands generate an SQLException. In Lesson 5, “Cloudscape Database Applications”, you will learn how to catch this particular exception to avoid printing an error message.

Examine the System Directory and Information Log

Your application has now created and interacted with a Cloudscape database.

Examine the contents of your system directory. The directory should now include:

• The information log, cloudscape.LOG
  
  Open up this file and examine the contents. It should contain text similar the following:
  
  Sun Oct 04 13:28:00 PDT 1998:
  Booting Cloudscape version 3.5.1: instance 80000000-00d3-4d2c-ca4d-000a0a0b4300
  on database at directory D:\my_system_directory\HelloWorldDB
  
  Sun Oct 04 13:28:07 PDT 1998:
  Shutting down cloudscape instance 80000000-00d3-4d2c-ca4d-000a0a0b4300
  
  As you can see, the log shows more than just errors—it shows information about system booting and shutdown.

• A directory called HelloWorldDB (this is where Cloudscape stores the data, folks!)
  
  Open up this directory and look at what’s inside; it contains two subdirectories and two files. You can learn more about what those are in the Cloudscape documentation set.

Building the Database toursDB

Now that you know the basics of how a Cloudscape application works, you will run a program from the JBMSTours package to create the toursDB sample database described in Chapter 1, “Overview”.

(The installation includes a pre-built version of toursDB, which is in the /demo/databases directory under the cloudscape base directory. But for the purposes of this tutorial, do not use that version.)
Cloudscape Basics and the Sample Database

Run JBMSSTours.CreateToursDB

*JBMSSTours.CreateToursDB* is a Java application that creates the *toursDB* database (and connects to it), creates all the database’s dictionary objects, such as tables, indexes, and views, and then populates the core tables with data. (You will examine the dictionary objects and the data in the next chapter.)

Compiled classes are included in the *demo/programs/tours/JBMSSTours* directory. You will also find the source files there.

To run *JBMSSTours.CreateToursDB*:

1. Return to the first command prompt window that you opened. You should be in the *your_tutorial_home* directory.

2. If you are running on a UNIX platform, you may need to relax your file descriptors limit. For more information, see Appendix A, “Troubleshooting Common Problems”.

3. Run the program, using the *-D* parameter to set *cloudscape.system.home*:

   ```
   java -Dcloudscape.system.home=your_tutorial_system
   JBMSSTours.CreateToursDB
   ```

   The program takes a few minutes to run, depending on your configuration. While it runs, it prints messages about the tasks it is performing, such as building tables, constructing objects, and inserting data. When it is complete, it displays the following:

   ```
   Committed transaction and closed connection
   Database shut down normally
   CreateToursDB finished
   ```

   Congratulations! You have successfully built two databases, including the *toursDB* sample database. You are ready to move on to the next chapter.
### Table 7 Tutorial Labs Reference Sheet

- **yourClassPath**
  The complete list of paths to directories and zip files needed to run the applications in this tutorial. See “Getting Acquainted with Class Path” on page 2-2.

- **classPathScript**
  The script text file you copied from scripts to your_tutorial_home and modified to run as a script that sets your class path. See “Customize Your setclasspath Script” on page 2-8. To run it, see “Running the Script” on page 2-8.

- **your_tutorial_system**
  The path to your system directory, the directory that contains your databases. In this tutorial, it is the tutorial_system directory that you created in “Getting to Know Your System Directory” on page 2-6. This directory will contain the databases that you create in this tutorial, as well as the information log and properties file.

  ```
  -Dcloudscape.system.home = c:\tutorial_system
  ```

- **demo/programs/tours/JBMSTours**
  The default installation includes a directory called demo. Inside this directory is a subdirectory called programs/tours, which contains the JBMSTours package. Source files are located there, as well as the javadoc subdirectory.

- **javadoc**
  Javadoc is automatically generated HTML documentation about Java classes. Javadoc for the JBMSTours application package are in the subdirectory javadoc within demo/programs/tours. For example, javadoc for the JBMSTours.serializabletypes.City class is in demo/programs/tours/javadoc/JBMSTours.serializabletypes.City.html

- **scripts**
  This directory contains scripts and some example Java source files.

  *scripts* is a subdirectory in demo/programs/tours (in the cloudscape base directory).

  Copy these files to your_tutorial_home; don’t alter the original files.

- **your_tutorial_home**
  This directory should be your home base when working in a command window. In other words, execute all commands from this directory unless otherwise directed.

  Copy example scripts and source files from scripts to this directory. Also contains the classPathScript that you create.
3  Using Cloudview

Concepts covered:

- *Relational databases*
  “Tables in a Relational Database” on page 3-2
- *Dictionary objects*
  “Working with Cloudview to Create Dictionary Objects” on page 3-3
- *SQL data types*
  “Examining Tables and Data in toursDB with Cloudview” on page 3-9

Tasks:

- “Start Cloudview in the Home Directory” on page 3-3
- “Create the Sayings Table” on page 3-4
- “Create the Responses Table” on page 3-5
- “Define a Primary Key” on page 3-6
- “Define a Foreign Key” on page 3-7
- “Enter Data Graphically and with an SQL Statement” on page 3-7
- “Open and Execute an SQL Batch File” on page 3-8
- “Browse the Data in the Sayings Table” on page 3-8
- “Open a Connection to toursDB” on page 3-9
- “Browse the Definition of the AIRLINES Table” on page 3-9
- “Browse the Data in the COUNTRIES Table” on page 3-10

**NOTE:** You can do this lesson as a stand-alone demo. To do that, first do Lesson 2.
Tables in a Relational Database

A relational database stores information in two-dimensional tables consisting of rows and columns, as illustrated in Figure 3-1. Tables contain rows of data consisting of columns of simple data types such as strings of characters, numeric data types, dates, times, and a few others.

A relational database usually consists of more than one table. Tables are related to each other through keys, columns that define uniqueness and references between tables. Access to data that spans more than one table is called a join.

In the last lesson, you created a database in a small Java program. The command to create a database, as you recall, was a call to the java.sql.DriverManager.

However, you did not store any data in the database. In order to store data, you need to define tables for storing data.

Cloudscape provides a graphical application that makes it easy to create tables and other dictionary objects. This application, called Cloudview, does the work of sending the SQL-J commands that create the tables to Cloudscape. SQL-J is the name of Cloudscape’s dialect of SQL.

In this lesson, you will start Cloudview and use it to create and examine tables.
Working with Cloudview to Create Dictionary Objects

In this project, you will create two tables in the HelloWorldDB database, store data, retrieve data, and update data.

Start Cloudview in the Home Directory

The Java class name of Cloudview is COM.cloudscape.tools.cview.

1. Open a command window and change directories to your_tutorial_home.
2. Run your classPathScript in that directory.
3. Run the program, using the -D parameter to set cloudscape.system.home:

   java -Dcloudscape.system.home=your_tutorial_system
   COM.cloudscape.tools.cview

The Main window appears:

![Cloudview Main Window](image)

Figure 3-3 The Main window
Using Cloudview

Connect to HelloWorldDB

1 To open a connection to the HelloWorldDB database, choose Open by Name from the File menu.

2 Type HelloWorldDB, which will connect to a database of that name in the system directory.

Quiz: Where is the system directory?

Answer: Wherever cloudscape.system.home points. If not set, it points to the current directory.

3 Click OK to connect.

The database screen appears. The left-hand hierarchy window shows the system and database hierarchy.
Underneath the system icon, all the open databases appear. Only one database is now open, HelloWorldDB.
Underneath the database icon are some icons representing other database objects such as tables, views, aliases, and more. Since HelloWorldDB has no database objects yet, no entities appear under these icons.
You will create a table in the next task.

Create the Sayings Table

1 Select the HelloWorldDB icon.

2 Choose New->Table from the Edit menu.
The Table window appears in the right-hand window.

3 Type Sayings in the Name box to name the table Sayings.
Tables that you create are case-insensitive; they are always stored as uppercase unless you choose the case-sensitive option (see the Preferences tab).
The table will have two columns: Number and Saying.

4 Type Number in the Name box in the Columns section.
Select the Type box of the current row to get a drop-down list of available data types.

Choose SMALLINT as the data type.

Select No from the drop-down list for Nullable. (This column must always contain a value).

Click the + icon to create a new column.

Type Saying in the Name box.

Select the Type box to get a drop-down list of available data types.

Choose VARCHAR as the data type.

Type 50 in the Length box.

The column can store varying-length strings of up to 50 characters.

Click OK to save the table definition to the database.

**Create the Responses Table**

1 Select the HelloWorldDB icon in the left-hand hierarchy window.

2 Choose New->Table from the Edit menu.

   The Table window appears in the right-hand window.

3 Type Responses in the Name box to name the table Responses.

   The table will have two columns: Number and Response.

4 Type Number in the Name box in the Columns section.

5 Select the Type box to get a drop-down list of available data types.

6 Choose SMALLINT as the data type.
7 Select No from the drop-down list for Nullable. (This column must always contain a value).
8 Click the + icon to create a new column.
9 Type Response in the Name box.
10 Select the Type box to get a drop-down list of available data types.
11 Choose VARCHAR as the data type.
12 Type 50 in the Length box.
   The column can store strings of up to 50 characters in length.
13 Click OK to save the table definition to the database.
   Two new icons should now appear under the Tables icon in the left-hand hierarchy window for the two new tables.

Define a Primary Key

A table should have a primary key. A primary key defines that column or columns that uniquely identify a row in a table. When you define a primary key on a column or column, the values in that column or columns must be unique; no two rows can have the same values in that column or columns.

In the Sayings table, the Number column should be the primary key.

1 Select the Sayings table icon, and click the plus sign to the left of the table.
2 Select the keys icon.
3 Choose New->Key from the Edit menu.
4 Type Sayings_Pk in the Name box.
5 The Number column should already be selected as the candidate for primary key column, so click OK.
Define a Foreign Key

The Number column in the Responses table corresponds to the Number column in the Sayings table. The Responses table should have a foreign key that enforces that relationship.

1. Select the Responses table icon, and click the plus sign to the left of the table.
2. Select the keys icon.
3. Choose New->Key from the Edit menu.
4. Type Responses_fk in the Name box.
5. The Number column should already be selected as the candidate for foreign key column.
6. Choose Foreign Key from the drop-down list of key types.
7. Choose Sayings from the drop-down list in the Foreign Tables field.
8. Its primary key (Sayings_Pk) should already be selected as the correct primary key to reference.
9. Click OK.

Enter Data Graphically and with an SQL Statement

Cloudview provides a graphical way to enter data.

1. Select the SAYINGS icon in the left-hand hierarchy window.
2. Click the Data tab.
   No rows appear, because the table is empty.
3. Click the + icon to add a new row.
4. Type 1 in the Number column and Hello, World! in the Saying column.
5. Click the Save icon to save your changes. The Save icon looks like this:

You can also enter data with an SQL statement. You will do that next.

6. Click the HelloWorldDB icon in the left-hand hierarchy window.
   The database screen appears in the right-hand window. The database screen includes an SQL window.
7. Type the following SQL-J statement in the SQL window:
Using Cloudview

```
INSERT INTO SAYINGS VALUES (2, 'You can’t have your cake')
```

If you are doing this tutorial on-line, you can cut the statement directly from this document and paste it into the window. This statement will insert the integer 2 and the words “You can’t have your cake” into the first row of the database.

**NOTE:** You must enclose strings in single quotation marks in the SQL-J language. In addition, since a single quotation mark (which is the same as an apostrophe) delimits the statement, you must “escape” the apostrophe within the string with another single quotation mark.

SQL-J commands are case-insensitive. That means that you could type “INSERT INTO” and “VALUES” in uppercase, lowercase, or any mix of the two. For clarity and consistency, this book shows SQL-J commands in ALL CAPS.

8 Click the lightning bolt to execute.
After the statement execution has completed, information about how long it took to compile and execute appears at the bottom of the window.

Open and Execute an SQL Batch File

Cloudview allows you to open and execute text files that contain one or more SQL-J statements. In this task, you will execute the `world.sql` text file to enter more data into the database.

1 Click the Script icon above the SQL window and choose Open.

2 Navigate to your home directory, then select `world.sql`, then click Open.
A batch of SQL-J statements appears in the window. A batch is a collection of statements. When executing more than one statement at once, you separate statements with semicolons.

3 Click the lightning bolt to execute.

4 You can click the left arrow to scroll back to the first statement you executed, if you wish. Click the right arrow to go back to the most recent statement.

Browse the Data in the Sayings Table

Cloudview provides a graphical data browser for viewing data in a table—it is the same window you used to enter your first row.

1 Select the Sayings table icon in the left-hand hierarchy window.
2 Click the Data tab in the right-hand window.
   The data you just entered appears in the data grid.
   You may need to enlarge the width of the Sayings column to view all the data.
3 Select the Responses table icon to view the data in the Responses table.

Examining Tables and Data in toursDB with Cloudview

Most databases are more complex than HelloWorldDB, of course. toursDB is a good example of a more complex database. In this project, you will open a connection to toursDB and then examine some of its tables and data.

Open a Connection to toursDB

1 To open a connection to the toursDB database, choose Open by Name from the File menu.
2 Type toursDB and click OK to connect.
   The toursDB database icon now appears in the left-hand hierarchy window.
3 Click the + to the left of the Table icon to display its tables.

The names of all user tables appear in the list.

Browse the Definition of the AIRLINES Table

1 Select the AIRLINES table.
2 Select the Table tab.
   Information about the AIRLINES table definition appears in the right-hand window.
3 Widen the Name and Type fields so that they display the full values.
Using Cloudview

This table stores data of four different types:

- **CHAR**
  A fixed-length string data type. The strings in the AIRLINE column are always two characters long. When strings are always a fixed-length, it is more efficient to store them in CHAR columns. Cloudscape always allocates the right amount of space for them.

- **VARCHAR**
  A variable-length string data type. The strings in the AIRLINE_FULL column are from 1 to 24 characters long.

- **DOUBLE PRECISION**
  A floating-point number up to 8 bytes.

- **INT**
  An integer.

### Browse the Data in the COUNTRIES Table

1. Select the COUNTRIES table icon.
2. Select the Data tab.
   Scroll through the data.

   **Quiz:** Can you guess the data types in this table?

3. Click the Table tab to get the table definition view.
   COUNTRY_ISO_CODE column is of the type CHAR, with a length of 2, and the other two columns are of type VARCHAR. Those columns have a maximum length of 26 characters specified (no values are longer than 26 characters).

Learning Cloudscape
Disconnect, Exit (and Shut Down Cloudscape)

If you are going straight to Lesson 4, “Objects and SQL”, do not do this task.

1 Disconnect from the *toursDB* database by selecting the *toursDB* icon and choosing Close from the Edit menu.

2 Disconnect from the *HelloWorldDB* database by selecting the *HelloWorldDB* icon and choosing Close from the Edit menu.

3 Shut down Cloudview by exiting the application.
   Shutting down Cloudview issues a shutdown command to Cloudscape in an embedded environment.

<table>
<thead>
<tr>
<th>Quiz: What does a shutdown command do?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Answer:</strong> It ensures that all changes are written from the database log (a temporary holding place for transactions) to the actual database. If you do not shut down Cloudscape, the next time it connects to a database it takes the time to run recovery on the database and slows down connection time.</td>
</tr>
</tbody>
</table>
4 Objects and SQL

Concepts covered:

- **Browsing objects: the object inspector**
  “Using Cloudview to Work with Stored Objects” on page 4-2
- **Quick introduction to “Database-side” method execution**
  “Executing Methods from the Object Inspector” on page 4-3
- **Working with SQL**
  “Introducing . . . SQL!” on page 4-5
- **Getting to know the data in toursDB**
  “Querying toursDB” on page 4-9

Tasks:

- “Examine the Cities Table Definition” on page 4-2
- “Browse the Contents of the Cities Table” on page 4-2
- “Browse a City Object” on page 4-3
- “Execute an Object’s Methods” on page 4-3
- “View the Superclass’s Methods.” on page 4-4
- “Select Data from the Sayings Table” on page 4-5
- “Execute an SQL-J Statement with a WHERE Clause” on page 4-6
- “Execute Joins” on page 4-7
- “Execute Methods in SQL-J Statements” on page 4-8
- “Execute Queries in toursDB” on page 4-9
- “Use a WHERE Clause” on page 4-9
- “Use More Complex Search Conditions” on page 4-10
- “Join the Countries and Cities Tables” on page 4-10
- “Disconnect, Exit (and Shut Down Cloudscape)” on page 4-11
NOTE: You can do this lesson as a stand-alone demo. To do that, first do Lesson 2.

Using Cloudview to Work with Stored Objects

Cloudscape’s Java extensions allow Cloudscape to store more than just SQL-92 data types. It can store instances of Java classes, known as Java objects.

To store Java objects in a table, associate a column with their Java class. The Cities table stores JBMSTours.serializabletypes.City objects, instances of the class JBMSTours.serializabletypes.City.

Open the javadoc file for JBMSTours.serializabletypes.City or view the source file for JBMSTours.serializabletypes/City for information about this data type. Javadoc for JBMSTours is in the demo/programs/tours/javadoc directory.

Examine the Cities Table Definition

1. If you are not currently in Cloudview with an open connection to the toursDB database, start the program as described in “Start Cloudview in the Home Directory” on page 3-3. Then open a connection to toursDB as described in “Open a Connection to toursDB” on page 3-9. Select the toursDB icon in the left-hand hierarchy window. Click the “+” to the left of the icon to display the Table icon.

2. Click the “+” sign to the left of the Table icon to display the tables in toursDB.

3. Select the Cities table icon.

4. Select the Table tab.

5. Examine the column definitions in the Cities table. Note that the second column, called CITY, is a Java data type. The column is associated with the JBMSTours.serializabletypes.City Java class.

Browse the Contents of the Cities Table

1. Select the Data tab in the right-hand window to display the contents of the Cities table.
2 Scroll through the data.

*CITY_ID* and *COUNTRY_ISO_CODE* are SQL-92 data types, so you can see the complete contents by browsing this window. However, the *City* column contains Java objects. You cannot “see” those objects from this window. These objects contain several fields, which are not displayed. What you see is only the *toString* values of the objects.

### Browse a City Object

Cloudview offers a tool for viewing some of the data contained in objects: the Object Inspector.

1 Select *Santiago, Chile* in the *CITY* column (*CITY_ID* 40), then click the Magnifying Glass (inspect) button.

   The Object Inspector window appears.

2 Click the “+” to the left of the *JBMS*Tours.serializables.City icon to display its fields. If a field is public, its value is displayed.

   Some of the fields are public, and thus their values are displayed.

3 Leave the Object Inspector window open for the next section.

### Executing Methods from the Object Inspector

Without methods, an object is just plain old data. Cloudview’s Object Inspector lets you view an object’s methods (and the methods belonging to any of an object’s superclasses) and to execute methods that do not take any parameters.

### Execute an Object’s Methods

1 Select *CITY=JBMS*Tours.serializables.City under Selected Objects.
Cloudview allows you to execute methods with no parameters from this window by selecting the method name and clicking Execute. The methods defined in `JBMS.Tours.serializabletypes.City` appear in the right-hand window.

2. Select the `getAirport()` method and click Execute. Santiago’s airport code, “SCL,” appears in the bottom right-hand window.

3. Select any other method that takes no argument, and click Execute.

**View the Superclass’s Methods.**

`JBMS.Tours.serializabletypes.City` is a subclass of `java.lang.Object`. You can view (and execute, if appropriate) methods defined in the superclass.

1. In the pop-up menu in the top right-hand window, select Object.


2. Close the Object Inspector window.
Introducing . . . SQL!

In Lesson 3, “Using Cloudview”, you got a rude introduction to SQL and learned about the INSERT command. This section is a proper, more polite introduction to the language used to interact with the database management software.

You use SQL to:

- retrieve data from a database, to store new data, and to modify existing data
  These commands are sometimes called data modification language, or DML. They include queries, inserts, updates, and deletes.
- to define or modify dictionary objects such as tables, views, and indexes
  These commands are sometimes called data definition language, or DDL

Queries

An SQL statement that retrieves data is called a query; the basic keyword in a query is the word SELECT. SQL grammar allows you to specify which tables and, within tables, which columns you want to retrieve data from.

Select Data from the Sayings Table

In Lesson 3, “Using Cloudview”, you used Cloudview to look at data in the Sayings and Responses tables.

In this task, you will execute an SQL-J statement to query the Sayings and Responses tables.

1 Open a connection to the HelloWorldDB if you have not already done so (follow the instructions in Lesson 3, “Using Cloudview”).
2 Click the HelloWorldDB icon in the left-hand hierarchy window.
   The database screen appears in the right-hand window.
3 Type or paste the following SQL-J statement into the SQL window:
   
   SELECT * FROM Sayings
   
   In SQL, an asterisk (*) stands for “all columns.” Selecting * from a table selects the data in all the columns.
4 Click the lightning bolt to execute.
   The results appear in the bottom window.

Cloudscape Version 3.0
5 Edit the SQL-J statement as follows—replace * with saying:

```sql
SELECT saying FROM Sayings
```

Specifying a column name or names allows you to retrieve data from specific columns only.

6 Click the lightning bolt to execute.
The results appear in the bottom window.

---

**WHERE Clauses**

You can also specify which rows to retrieve by constructing a search argument. A search argument consists of a clause to the SQL statement beginning with the keyword WHERE and is called a WHERE clause. A WHERE clause specifies conditions that the values stored in rows must meet in order to be retrieved.

---

**Execute an SQL-J Statement with a WHERE Clause**

Look for the saying whose NUMBER value is 2.

1 Type or paste in the following SQL-J statement:

```sql
SELECT * FROM Sayings WHERE number = 2
```

2 Click the lightning bolt to execute.
The results appear in the bottom window. Only the row for which the NUMBER value is 2 appears in the window.

---

**Joins**

A join is a query in which you select data from related tables. One way to specify how the tables are related to one another is to use the JOIN . . . USING commands.
JOIN means to join the two tables specified; USING specifies the column to use in joining the tables.

The two tables in the HelloWorldDB are related to each other by the NUMBER column (as you know, because you already created a foreign-key constraint on that column in the Responses table).

## Execute Joins

Look for all sayings and their corresponding responses.

1. Type or paste in the following SQL-J statement:

   ```sql
   SELECT SAYING, RESPONSE FROM Sayings JOIN Responses
   USING (NUMBER)
   ```

   This statement joins the two tables based on the NUMBER column, which has the same name in both tables. The value of the NUMBER column is what links a saying to its corresponding responses.

2. Click the lightning bolt to execute.

   The results appear in the bottom window. Each saying appears with its related responses. A saying appears once for each response it has. Since in this database each saying has two responses, each saying appears twice in the result set, even though a particular saying appears only once in the Sayings table. Only those sayings with responses are shown.

3. Resize the Saying and Response columns as needed to view the entire text.

   You can combine search arguments with a join specification. Edit the current SQL-J statement so that it returns only those rows for which the value in the NUMBER column is 3.

4. Add the following WHERE clause to the statement:

   ```sql
   WHERE Sayings.NUMBER = 3
   ```

   When specifying a column in a WHERE clause for which the name is ambiguous, you must specify the table name along with the column name. Specifying the NUMBER column is ambiguous, because there is a NUMBER column in both Sayings and Responses.

5. Click the lightning bolt to execute.

   You can also execute a join in a statement without the JOIN keyword. Instead, you use the WHERE clause to specify which columns must be related.
6 Execute the following statement:

```sql
SELECT SAYING, RESPONSE FROM Sayings, Responses
WHERE SAYINGS.NUMBER = RESPONSES.NUMBER
```

You should get the same results as in the first query in this section.

**NOTE:** In the current release, Cloudscape can often execute such a statement faster than one using the JOIN keyword.

### Execute Methods in SQL-J Statements

In “Executing Methods from the Object Inspector” on page 4-3, you executed methods on objects stored in the database using the Object Inspector window. You can also execute methods within a SQL-J statement.

Even though there are no Java data types in the `Sayings` or `Responses` tables, we can execute Java methods on them because VARCHAR values are automatically mapped to the `java.lang.String` class. SMALLINT values are automatically mapped to the `java.lang.Short` class. You can execute the methods of those classes on the values stored in the `Sayings` and `Responses` tables.

To execute a method on an object stored in a column, you add the method call with any parameters to the column name. A dot separates the column name from the method call. Cloudscape executes the methods once for each value returned in the result set for the particular column.

In this task, you will get all the sayings and convert them to uppercase.

**NOTE:** Remember that you delimit strings within SQL-J statements with single quotation marks.

1 Type or paste in the following SQL-J statement in the SQL window and then execute it:

```sql
SELECT SAYING.toUpperCase() FROM Sayings
```

Since `toUpperCase` is a Java method, its name is case-sensitive and must be typed exactly as it appears in the class definition. Results appear in the bottom window.

2 Execute the following weird SQL-J statement in the SQL window:
**Querying toursDB**

The *toursDB* database is a richer database than *HelloWorldDB* for learning more complex SQL-J.

Return to your open *toursDB* connection by selecting the *toursDB* icon.

**Execute Queries in toursDB**

Show all the data in the *Countries* table.

1. Select the *toursDB* database icon in the left-hand window so that you can execute SQL statements against that database.

2. Type or paste in the following simple query in the SQL window:

   ```sql
   SELECT * FROM Countries
   ```

   Table and column names in *toursDB* are not case-sensitive, because they were not created as delimited identifiers by Cloudview. That means that you can refer to the table and column names using mixed upper- and lowercase, as long as you do not delimit them with double quotation marks.

3. Click the lightning bolt to execute.

   The results appear in the bottom right-hand window.

**Use a WHERE Clause**

Find those flights that originate from JFK, the New York City airport.

1. Type or paste in the following query:
Use More Complex Search Conditions

You can add clauses to a WHERE clause using the keyword AND. Find flights that originate from JFK and that are longer than five hours:

1. Try the following query:

   ```sql
   SELECT * FROM Flights
   WHERE orig_airport = 'JFK'
   AND flying_time > 5
   ```

2. Narrow the search even further to find only those flights over 4000 miles:

   ```sql
   SELECT * FROM Flights
   WHERE orig_airport = 'JFK'
   AND flying_time > 5
   AND miles > 4000
   ```

Join the Countries and Cities Tables

Find out which cities are in which countries, using a join.

1. Try the following query:

   ```sql
   SELECT Country, City FROM Countries JOIN Cities
   ON Countries.country_ISO_code = Cities.country_ISO_code
   ```

2. Click the lightning bolt to execute.

   Results appear in the bottom window.

   This kind of join is called an inner join. For each table, it returns only rows for which there is joined data in the other table. For example, countries that have no corresponding cities do not appear in the result set.

   A country has one row for each city, so some countries appear in more than one row.

   A join that includes rows that have no corresponding values in the joined table is called an outer join.

   To see all countries, including those that have no corresponding cities, you need to modify this query to make it a left outer join. A left outer join means
that the left (or first) table in the query is the one for which the extra rows are preserved.

3 Try the following query:

   SELECT Country, City FROM Countries LEFT OUTER JOIN Cities
   ON Countries.country_ISO_code = Cities.country_ISO_code

4 Click the lightning bolt to execute.

   Results appear in the bottom window. Scroll through the results if you like. Notice that some countries, such as Armenia and Angola, do not have any cities associated with them in this database, so the city column is blank (null) for their rows.

**Disconnect, Exit (and Shut Down Cloudscape)**

1 Disconnect from the database by choosing Close Database from the Edit menu on the toursDB and HelloWorldDB icons.

2 Shut down Cloudview by exiting the application.

   Shutting down Cloudview issues a shutdown command to Cloudscape in an embedded environment. (In a client/server environment, it does not shut down Cloudscape.)
5 Cloudscape Database Applications

Concepts covered:

- *Embedded vs. server environment*
  “Cloudscape Deployment Options” on page 5-1
- *JDBC classes and methods*
  “Working with the JDBC Interface” on page 5-6
- *SQLExceptions*
  “Handling SQLExceptions” on page 5-9
- *More programs*
  “Running the BuildATour Application” on page 5-14

Tasks:

- “Run and Examine HelloWorldApp.java” on page 5-7
- “Change the logSeverityLevel Property” on page 5-11
- “Compile, Run, and Examine HelloWorldExc” on page 5-11
- “Run JBMSTours.BuildATour” on page 5-14

Cloudscape Deployment Options

This section covers the possible deployment options for Cloudscape and how they differ.

- “What Does Embedded Mean?” on page 5-2
- “Cloudscape’s Server Frameworks” on page 5-3
What Does Embedded Mean?

When you deploy a Java application along with the Cloudscape engine and a copy of your database, Cloudscape is said to be embedded in the application. As you saw in Lesson 2, “Cloudscape Basics and the Sample Database”, the application makes a call to the Cloudscape JDBC driver to start up the local Cloudscape software. The application does not need any special network connections to access the database software. When embedded in a single-user application, the Cloudscape software appears to be an integral part of the application and does not require any action or configuration on the user’s part.

An application starts up Cloudscape running in an embedded environment by loading the Cloudscape driver:

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

An application connects to Cloudscape running in an embedded environment by specifying the database connection URL as an argument to a call to the `DriverManager`:

```java
Connection conn = DriverManager.getConnection("jdbc:cloudscape:toursDB");
```

After that, the application uses the methods of the `java.sql` API to execute SQL-J statements against the local Cloudscape database.
**NOTE:** An ODBC-JDBC bridge will be available on our support Web site. This driver runs on top of the JDBC driver and allows existing ODBC applications to run against Cloudscape.

**Application Servers and Embedded Cloudscape**

Applications that provide services to many users can, like single-user applications, run with the Cloudscape engine embedded in them; like single-user disconnected applications, they run inside the same JVM as Cloudscape and make calls to Cloudscape through the local JDBC driver.

Because Cloudscape runs embedded in the application, Cloudscape simplifies three-tier scenarios, eliminating network overhead between the application server and the DBMS.

Some third-party application servers may require a special driver on top of Cloudscape’s local JDBC driver. Check our support Web site for information about the drivers that are available.

**Cloudscape’s Server Frameworks**

Cloudscape also offers JDBC server frameworks, which provide JDBC and in some cases HTTP services to remote client applications. When Cloudscape runs inside
one of these frameworks, multiple users can access the same database or databases through the same server framework, which runs on a central server machine.

The two frameworks provided by Cloudscape are:

- **RmiJdbc**
  A customized version of a freeware product with a small footprint that comes with Cloudscape.

- **Cloudconnector**
  A full-featured JDBC and HTTP server available for purchase separately.

---

**Figure 5-3**  When Cloudscape runs inside a server framework, client JDBC applications run in different JVMs (and most likely different computers) from the server. The database is stored in the file system local to the server.

The system administrator starts up the server on the central machine.

Client applications do not start up Cloudscape, but instead make calls to the server’s client software. The server must already be running.

Client applications start up the server’s client software by loading the driver. The examples in this section are for use with the Cloudconnector framework:

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

An application connects to Cloudscape running in an embedded environment by specifying the Cloudconnector client database connection URL as an argument to a call to the DriverManager:
Connection conn = DriverManager.getConnection("jdbc:cloudscape:weblogic:toursDB");

After that, the application uses the methods of the java.sql API to execute SQL-J statements against the local Cloudscape database.

After connecting, client JDBC applications running against a server and JDBC applications running against Cloudscape in an embedded environment work the same—through the methods of the JDBC interface. The only exception is that client applications do not need to shut down Cloudscape.

For an example, see “JDBC Servers” on page 11-2.

A Cloudscape Synchronization System

Cloudscape applications running against Cloudscape in an embedded environment can use the Cloudscape synchronization technology. The synchronization-enabled version of Cloudscape, called Cloudsync, has all of Cloudscape’s features plus the synchronization technology.

Synchronization allows occasionally connected databases to connect to a central Cloudscape database, known as a source, via the HTTP protocol and to share and synchronize data.

When not connected, the remote application and Cloudscape software run exactly like an application running against Cloudscape in an embedded environment. The application starts the local JDBC driver, specifies the Cloudscape JDBC database connection URL, and interacts with the database through the methods of the JDBC interface. The synchronization technology handles updating data back and forth from the deployed databases and the server.

When an embedded database uses synchronization, it is known as a target database. The source database runs inside a server product. Such a configuration supports the multiple HTTP connections from the target databases plus connections from JDBC client applications connected over the network.
Working with the JDBC Interface

The preceding section discussed the different ways of deploying the Cloudscape products. Despite their differences, these deployment options have something in common: namely, the JDBC interface. All Cloudscape applications use the JDBC interface to interact with Cloudscape, whether it is embedded or a server.

**NOTE:** Application servers may provide a connectivity option other than JDBC to their client applications. For example, BEA WebLogic’s application server provides a dbKona layer above JDBC that client applications can use instead of JDBC. A servlet application enables Web browsers to be non-JDBC client applications. However, the application server itself must use JDBC for interacting with Cloudscape. Throughout the Cloudscape documentation, the term Cloudscape application refers to an application that interacts with Cloudscape using the JDBC interface.

A Java application talks to a database using the classes and methods of the `java.sql` package.

This section does not go into great detail about the JDBC interface. Instead, it offers a brief overview and teaches you some highlights. To develop Cloudscape applications successfully, you will need to learn the JDBC interface as well as SQL-J.

**Applications and Queries: Statements and ResultSets**

When you worked with Cloudview, you used its SQL window to send SQL-J statements to Cloudscape. A Java application typically uses a *Statement* object or a *PreparedStatement* object to send SQL-J statements to Cloudscape.

A *Statement* object lets you execute a static SQL-J statement and obtain its results. You create a *Statement* object using the `createStatement` method of the *Connection* class:

```
Statement stmt = conn.createStatement();
```

After you create a *Statement* object, invoke one of its execute methods to send an SQL-J statement to Cloudscape. When you invoke it, supply an SQL-J statement as a string argument to the method. If the SQL-J statement is a query (returns data) and does not modify a schema or data, use the `executeQuery` method; if it modifies schema or data, use the `executeUpdate` method. (Use `execute` for those situations in which you don’t know in advance what the statement does.)

Queries return results; when executing queries, you assign the results to a *ResultSet* object.

**Run and Examine HelloWorldApp.java**

*HelloWorldApp.java* is a simple Java application that interacts with *HelloWorldDB*—it queries a table and prints the results.

1. Open a command window and change directories to `your_tutorial_home`.
2. Run your `setclasspath` script.

```
javac HelloWorldApp.java
```
4. Run the program, specifying the system directory with the `-D` parameter:
The program displays output as it runs. The output should look familiar to you; it is the contents of the *HelloWorldDB* database.

5 Open the *HelloWorldApp.java* source file. The contents of the file are also shown in Figure 5-5.

```java
import java.sql.*;
public class HelloWorldApp {
    public static void main(String[] args) {
        try {
            Class.forName("COM.cloudscape.core.JDBCDriver").newInstance();
            System.out.println("Loaded the cloudscape JDBC driver. Hello, World!");
            Connection conn = DriverManager.getConnection("jdbc:cloudscape:HelloWorldDB");
            System.out.println("Connected to HelloWorldDB database");
            Statement s = conn.createStatement();
            ResultSet rs = s.executeQuery("SELECT saying, response FROM Sayings JOIN Responses USING (number)");
            String saying, response;
            while (rs.next()) {
                saying = rs.getString(1);
                response = rs.getString(2);
                System.out.println(saying + " " + response);
            }
            rs.close();
            s.close();
            conn.close();
        }
        catch (Throwable e) {
            System.out.println("exception thrown");
            System.out.println(e);
        }
    }
}
```

You have already learned about loading the driver and connecting.

You have already learned about loading the driver and connecting.

Creating a statement against a connection will allow you to execute SQL statements.

Assign the results of the query execution to a ResultSet object.

Step through the rows in the ResultSet to access the data returned.

Explicitly close objects.

**Figure 5-5**  *HelloWorldApp*

The program loads the Cloudscape JDBC driver and gets a connection to *HelloWorldDB*. (You learned how to do this in Lesson 2, “Cloudscape Basics and the Sample Database”.)

Then the program creates a *Statement* object against the connection.
The program executes the `Statement`, assigning the results to a `ResultSet` object. The application then steps through the rows in the `ResultSet`. When in a row, it accesses the contents of a particular column using the `getString` method and specifying the column number (the order in which the column appears in the query) as an argument. JDBC begins counting at 1, not 0. There are different methods for different data types—an integer data type would require a `getInt` method, and so on. The `SAYING` column appears first in the query, so `getString(1)` retrieves the value in the `SAYING` column for each row. 1 refers to the first column in the result.

**Handling SQLExceptions**

`HelloWorldApp` lacks two basic steps to make it a real embedded Cloudscape application: `SQLException` handling and shutting down Cloudscape. As you learned in Lesson 2, “Cloudscape Basics and the Sample Database”, it is important for an application to shut down Cloudscape before exiting the JVM so that Cloudscape can issue a checkpoint.

**Quiz:** Do Cloudscape client/server applications shut down Cloudscape?

**Answer:** No. A server by definition runs continuously. Client applications connecting to a server should end by disconnecting only, not by shutting down.

Methods that access a Cloudscape database can throw `SQLExceptions` if something goes wrong during database access. Applications should handle `SQLExceptions` to get the details. For example, if the SQL-J statement specifies columns that do not exist, an `SQLException` is thrown.
Cloudscape Database Applications

Figure 5-6 How HelloWorldApp needs to be modified

The catch block of HelloWorldApp should treat exceptions in a more sophisticated manner and thus make HelloWorldApp a real Cloudscape application.

HelloWorldExc is a duplicate of HelloWorldApp, except that it makes the additions we identified in Figure 5-6:

- It processes SQLExceptions.
- It introduces a mistake in the SQL-J statement so that when you run it, the program displays error information from the SQLException.
- It shuts down Cloudscape properly.

Learning Cloudscape
Change the logSeverityLevel Property

You have already been working with one Cloudscape property, which is a configuration parameter for a Cloudscape database or application session. You have been setting the `cloudscape.system.home` property as a command-line option to the JVM every time you have run an application.

You can also set a property in a file called `cloudscape.properties`. This file is not included with the product; you must create it and edit it yourself. This file is one of the files from the `/demo/programs/tours/scripts` directory that you copied into your home or working directory.

In this task you will set the value of the `cloudscape.stream.error.logSeverityLevel` property so that you can see all errors written to the information log, regardless of their severity.

1. Copy the `cloudscape.properties` file into the `your_tutorial_system` directory.
2. Open the `cloudscape.properties` file in the `your_tutorial_system` directory. The file already sets one property:
   
   ```
   cloudscape.infolog.append=true
   ```
   
   This setting means that every time Cloudscape starts up, it appends to the information log (`cloudscape.LOG`) instead of overwriting it.

3. Set the value of `cloudscape.stream.error.logSeverityLevel` to 0 by adding the following line at the end of the file:
   
   ```
   cloudscape.stream.error.logSeverityLevel=0
   ```

4. Add a carriage return after the line, and save the file.

   This value means that Cloudscape will write errors of any severity level to the log. The error in `HelloWorldExc` is of a minor severity and so would not be written to the log using the default value for this property.

Compile, Run, and Examine HelloWorldExc

1. Return to your open command window, set to the `your_tutorial_home` directory.
2. Compile the `HelloWorldExc.java` file.
   
   The errors in the SQL-J statement are not detected by the compiler (the error will be detected by Cloudscape at runtime), so it should compile correctly.
Cloudscape Database Applications

3 Run the program from the your_tutorial_home directory:

```java
java -Dcloudscape.system.home=your_tutorial_system
        HelloWorldExc
```

The program displays information about the error as it runs. Some of this information may also be written to the information log.

**Quiz:** Do you remember where the information log is? (See “Examine the System Directory and Information Log” on page 2-17.)

4 Open and examine the HelloWorldExc.java source file.

To shut down Cloudscape, the program includes the shutdown command in a separate try block.

**Quiz:** You issue the shutdown command as an attribute to a database connection URL in a call to the DriverManager.getConnection method. What else can you do with this method?

**Answer:** Connect to databases and create new ones.

A successful shutdown command always raises an SQLException in Cloudscape. This may seem counterintuitive, but it does this to let you know that no connection is available. The catch block checks that the SQLException was issued correctly but does not process it.

```java
try {
    boolean gotSQLExc = false;
    try {
        DriverManager.getConnection("jdbc:cloudscape:;shutdown=true");
    } catch (SQLException se) {
        gotSQLExc = true;
    }
    if (!gotSQLExc)
        System.out.println("Database did not shut down normally");
    else
        System.out.println("Database shut down normally");
}
```

**Figure 5-7** Catching the SQLException

The main catch block catches exceptions, as shown in Figure 5-8. If an exception is an instance of an SQLException, it processes the exception to display information about it.
SQLExceptions return information about the SQLState and the ErrorCode. SQLStates are standard, ISO-defined codes describing the nature of the error. ErrorCodes are vendor-specific codes. In Cloudscape, these codes describe the severity of the error.

In addition, the JDBC specification states that a single error can generate more than one SQLException; an SQLException object can be “chained.” Use a loop and the getNextException method to process all SQLExceptions in the chain.

Cloudscape SQLExceptions use SQLState class codes starting with X. Cloudscape returns standard SQLState values or exceptions where appropriate.

**Closing Statements and ResultSets**

HelloWorldExc explicitly closes Connections, Statements, and ResultSets when they are no longer needed. Connections to Cloudscape are resources external to an application, and the garbage collector may not close them automatically.
Running the BuildATour Application

BuildATour is the so-called front-end user application for the JBMSTours company. We refer to it as “so-called” because it doesn’t really have a user interface. For the sake of simplicity, it automatically provides some data that a user might enter, then generates a customized tour based on information from the database and stores some information. The application illustrates how an application can interact with Cloudscape, not how an application interacts with a user.

In this section, you will run the BuildATour application, then look at parts of it to see more sophisticated use of statements.

Then you will take a quick look at the data entered by the application.

Run JBMSTours.BuildATour

1. Return to your open command window (the one you opened in “Run and Examine HelloWorldApp.java” on page 5-7).

2. Run the program, redirecting the output to the file buildout.txt:

   ```
   java -Dcloudscape.system.home=your_tutorial_system
   JBMSTours.BuildATour > buildout.txt
   ```

   When the program runs, it writes information about the tours it is building to the output file.
Run the same program again two or three times. It will build different tours and insert more data.

Open buildout.txt and examine the program’s output.

**Quiz:** Did BuildATour write new data to the database?

**Answer:** Yes. It added or updated data in several tables. Try any of the following queries from Cloudview:

```sql
SELECT Flights.flight_id,
    Flights.segment_number, flight_date,
    economy_seats_taken, business_seats_taken,
    firstclass_seats_taken
FROM Flights JOIN FlightAvailability
USING (flight_id, segment_number);

SELECT city.getName() AS CITY,
    hotels.hotel_name AS HOTEL, booking_date,
    rooms_taken
FROM Cities, Hotels, HotelAvailability
WHERE Cities.city_id = Hotels.city_id
AND Hotels.hotel_id =
    HotelAvailability.hotel_id;

SELECT Groups.group_id, People.person_id,
    person
FROM Groups, People, Groups_People
WHERE Groups.group_id =
    Groups_People.group_id
AND People.person_id =
    Groups_People.person_id;

SELECT *
FROM Groups;

SELECT customized_tour.toString()
FROM CustomizedTours;
```
6 Working with SQL-J’s Java Extensions

Concepts covered:

- *ij and SQL-J*
  “What’s the J in SQL-J? What’s ij?” on page 6-2
- *The City class practicum*
  “Meet the City Class” on page 6-3
- *Database-side methods*
  “Database-Side JDBC Methods” on page 6-10
- *VALUES expressions and java constructors*
  “VALUES Expression” on page 6-13
- *More on database-side methods*
  “Accessing Classes and Class Aliases” on page 6-14
- *Method aliases*
  “Static Method Aliases” on page 6-18

Tasks:

- “Examine and Execute CityTest” on page 6-3
- “Start ij” on page 6-6
- “Connect to toursDB” on page 6-6
- “Execute SQL-J Statements Referencing Stored City Objects” on page 6-6
- “Access a Field in an SQL-J Statement” on page 6-7
- “Execute More Complex SQL-J Statements” on page 6-7
- “Execute Methods in a Program as Application-Side Methods” on page 6-11
- “Execute Methods in Statements as Database-Side Methods” on page 6-11
Working with SQL-J's Java Extensions

- “Execute a Method in a VALUES Expression Using an Instance” on page 6-15
- “Execute a Method Using a Class, Not an Instance” on page 6-16
- “Work with Class Aliases” on page 6-16
- “Execute Some More Methods Using Class Aliases” on page 6-17
- “Use a Constructor” on page 6-18
- “Create a Method Alias” on page 6-19
- “Execute Method Aliases for Database-Side Methods” on page 6-19

Before you begin:

- Make sure you have completed all the tasks in Lessons 2 and 5.

What's the J in SQL-J? What's ij?

As you know, SQL-J is the name of the Cloudscape’s Java-enabled dialect and is pronounced “ess-queue-ell-jay.” (The J stands for Java.)

The SQL standard was designed for retrieving standard data types out of a database and inserting or modifying standard data types stored in a database. With SQL-J, you can integrate much of your Java syntax. As you saw in Lesson 4, “Objects and SQL”, you can store objects in a Cloudscape database and execute methods that belong to those objects. But Cloudscape can likewise access all Java classes that are available to the JVM in which it is running. For example, you can execute static methods of any class available to the JVM; if it is appropriate, you can also initialize an object of any available class and execute methods off it.

This lesson illustrates both possibilities: executing methods off objects stored in the database, and executing methods from a class.

ij (pronounced “eye-jay”) is the name of a tool provided by Cloudscape for running SQL-J statements entered interactively or from a script against a Cloudscape database. As you know, Cloudview provides an SQL window for executing SQL-J statements, among its many other features. ij’s sole purpose is to execute SQL-J statements, both singly and in batches. It provides more sophisticated control over the execution environment, and it also provides some commands that allow you to execute JDBC-level commands. For example, as you know, connecting to a database is done with a JDBC call. ij provides a command for connecting to a database.

The “i” in ij stands for interactive; the “j” stands for JDBC.
You can enter commands interactively or in scripts. In this lesson, you will execute the commands interactively.

**Meet the City Class**

In this section, you will be working with the `JBMSTours.serializabletypes.City` class. Open the javadoc or the Java source file for this class now. Scroll through the methods to start getting familiar with what these methods do. Javadoc for `JBMSTours` is in the `/demo/programs/tours/javadoc` subdirectory in the cloudscape base directory.

**Examine and Execute CityTest**

`CityTest.java`, which you should have copied into your working directory from the `scripts` directory, is a simple Java application that does not access the database. Instead, it creates two instances of the `City` class and executes some of its methods for one of those (it uses the other instances as a parameter to two method calls).

One of the methods it uses is `getDistanceFrom`. The `getDistanceFrom` method shows off the power of object programming. One of the fields in the `City` class is a `Location` object (you can look at the javadoc for `Location` to learn the details). The `Location` object defines the location in terms of latitude and longitude. The `getDistanceFrom` method uses a well known trigonometric formula to calculate the approximate distance between two locations.

In this task, you will run `CityTest`, which executes *application logic in the application*. In the next task, you will execute some of those same methods in SQL-J statements instead of from an application—executing *application logic in Cloudscape*. As you will see, there is not much difference.

1. Open and examine `CityTest.java`.
2. Open a new command window and change directories to the `your_tutorial_home` directory.
3. Run your `setclasspath` script.
4. Compile `CityTest.java`.
5. Execute this program:
java CityTest

Since this program does not use Cloudscape, you do not need to set the cloudscape.system.home variable as a -D parameter.

6 Notice the methods it executed and the results.

7 Keep the command window open. You will use it again in the next task.
Work with Stored City Objects

The toursDB database stores several City objects in the City table. You can execute some of the same methods executed by CityTest within an SQL-J statement. You will execute those statements within ij.

Notes About Starting ij

The name of the program ij is COM.cloudscape.tools.ij.

When you start ij, you will specify an ij property called ij.protocol. The ij.protocol consists of the “protocol” and “subprotocol” portions of the database connection URL you will be using. For example, the ij.protocol you have used throughout this tutorial is jdbc:cloudscape:. Users working in client/server mode use jdbc:cloudscape:weblogic:.

When you specify ij.protocol, ij automatically boots the appropriate JDBC driver. So if you specify jdbc:cloudscape:, ij automatically boots the embedded JDBC driver for you.

In addition, after you specify this property, ij allows you to provide a “short form” of a database connection URL to connect to a database. That is, instead of having to specify jdbc:cloudscape:HelloWorldDB to connect to HelloWorldDB (as you did in a Java application in Chapter 2, “Cloudscape Basics and the Sample Database”), you need only specify HelloWorldDB. You will use the short form of the database connection URL as an argument to the ij command Connect.

You specify the ij.protocol property on the command line when starting up ij, like this:

```
java -Dij.protocol=jdbc:cloudscape: COM.cloudscape.tools.ij
```

NOTE: The good news is that Cloudscape provides a batch file for you to make this easier. If you are working on a Windows or UNIX platform and you have the /bin subdirectory of your cloudscape base directory in your PATH, you can accomplish the same thing by merely typing ij. For example:

```
ij
```

The rest of this book assumes you don’t have this shortcut and shows you the long way, but use this shortcut if you have it.
Start ij

NOTE: ij commands are case-insensitive. ij command statements end with a semicolon.

1 Return to your open command window.

2 Start ij, specifying the system directory where the toursDB database lives with a \(-D\) parameter and the \(ij.protocol\) property:

```java
java -Dcloudscape.system.home=your_tutorial_system
-Dij.protocol=jdbc:cloudscape: COM.cloudscape.tools.ij
```

You should have an \(ij>\) prompt ready for you to enter SQL-J statements or ij commands.

\(ij>\)

Connect to toursDB

1 Establish a connection to toursDB:

```sql
connect 'toursDB';
```

The string you pass to the Connect command is the “short form” of embedded database connection URL.

The rest of the examples in this chapter do not include the \(ij>\) prompt, so that it is easier to cut and paste queries into the command window.

Execute SQL-J Statements Referencing Stored City Objects

Now that you are connected to the toursDB database, you can access the City table.

One of the methods that CityTest executed was the City class’s getName() method. CityTest executed that method on two City objects residing in memory. You can have Cloudscape execute that method on one, some, or all of the City objects stored in the table.

1 Execute the method on all the City objects (press Return after you type the statement):

```sql
SELECT city.getName() FROM CITIES;
```

\(city\) is the column name. That column happens to store instances of JBMS Tours.serializabletypes.City (the column could have been called something different).

Learning Cloudscape
2 Execute the method on those City objects whose city_id is greater than 50:

```java
SELECT city.getName() FROM CITIES
WHERE city_id > 50;
```

3 Execute the method only in the WHERE clause:

```java
SELECT city_id FROM CITIES
WHERE city.getName() = 'Santiago';
```

Executing a method in a WHERE clause allows you to refine your search conditions.

**Access a Field in an SQL-J Statement**

To access the field of a stored object in an SQL-J statement, you add the field name to the column name. The two-character combination “->” separates the field name from the column name.

For you to access a field, the field must be public.

The language field in the City class is public.

1 Execute the following statement:

```java
SELECT city->language FROM Cities;
```

2 Now do the same thing, but eliminate duplicates.

```java
SELECT DISTINCT city->language FROM Cities;
```

**Quiz:** Why can’t you access the airport field?

**Answer:** Because it is a private field.

**Execute More Complex SQL-J Statements**

CityTest executed two methods that took City objects as parameters. For example, the getDistanceFrom(City) method took a City object as a parameter. How can you execute those methods within an SQL-J statement?

Well, it’s possible but a little complicated.

One way to do it is to craft an SQL-J statement that returns a City object, and then to use that statement as the parameter to the method. It must return only one City.
object, not more. Since the `city_id` is a unique column, we specify a single `city_id` in the WHERE clause.

1. Execute the following SQL-J statement, which returns a single `City` (which happens to be Paris):

   ```sql-j
   SELECT City FROM Cities WHERE city_id = 35;
   ```

   That query should return the following:
   ```
   CITY
   ------------
   Paris, France
   ```

Let’s find how far each `City` in the database is from Paris. We can do that by passing in the above SQL-J statement as a parameter to the `getDistanceFrom(City)` method call. If the main SQL-J statement has no WHERE clause, Cloudscape evaluates the main expression for every row in the database.

2. Besides the city’s distance from Paris, also select the city name and give the method call a column name:

   ```sql-j
   SELECT city.getName(), city.getDistanceFrom(
       (SELECT city
            FROM Cities
            WHERE city_id = 35))
   AS MILES_FROM_PARIS
   FROM Cities;
   ```

   **Quiz:** Why does the SQL-J statement that returns the Paris `City` object have two sets of parentheses around it?
   
   **Answer:** One set of parentheses is for the method invocation. The second set of parentheses is to mark the statement as a subquery. Subqueries, which are SQL-J queries used to supply values within other queries, always require parentheses.

3. You can get Cloudscape to order the results, from nearest to farthest. Execute the following statement:

   ```sql-j
   SELECT city.getName(), city.getDistanceFrom(
       (SELECT city FROM Cities WHERE city_id = 35))
   AS MILES_FROM_PARIS
   ```
A DBMS’s ability to filter and sort records is one of the reasons to use a database instead of just storing data yourself.

*CityTest* executed the `getDistanceFrom` method only for Santiago.

You may recall that the output of *CityTest* was:

```
C:\tutorial_home>java CityTest
constructing city object for Santiago
constructing city object for Paris
The airport for Santiago is SCL
The distance between Santiago and Paris is 7244 miles
The current average temperature in Santiago is 66.5
The time difference between Santiago and Paris right now is -6.0 hours
```

4 Let’s execute the `getDistanceFrom` method only for Santiago (`city_id 40`, as you may recall) within an SQL-J statement and see if we get 7244 for the answer:

```
SELECT city.getDistanceFrom(
    (SELECT city FROM Cities WHERE city_id = 35)
) FROM Cities
WHERE city_id = 40;
```

You should indeed get 7244 as your answer.

5 Disconnect:

```
Disconnect;
```

6 Shut down the system by exiting:

```
exit;
```

The `ij exit` command automatically shuts down the system properly in an embedded environment.

This section and the last section showed how you can execute the same methods that you execute in a Java program in the database. When executed within an SQL-J statement, methods are called database-side methods.
Database-Side JDBC Methods

In the last section, you worked with a `getDistanceFrom` method that took a `City` object as a parameter. You constructed a subquery to “return” a City object to the method call. The method itself did not query the database.

The `JBMSTours` application programmer overloaded that method; the `City` class contains the method with a slightly different signature:

```
getDistanceFrom(Connection, int),
```

which does query the database. The second parameter to this method is the city’s id, which is an `int`. This method takes the city code, uses the connection (which is the first parameter) to query the database for the corresponding `City` object, then calculates the distance between the Cities and returns that value.

Since the method takes an `int` instead of a `City` object, we don’t need to do anything fancy to construct an object to pass to it. We can use a literal integer (in this case, 35, Paris’s city code). However, we will need to do something special to pass it a `Connection` object.

`getDistanceFrom(Connection, int)` accesses the `toursDB` database; a method that accesses the database when called within a SQL-J statement is called a database-side JDBC method, because it uses JDBC calls to execute SQL-J statements. A database-side JDBC method needs a `Connection` object. In most cases, the method uses the same `Connection` as the SQL-J statement that called it; such a connection is called a nested `Connection`. In Cloudscape, this can be handled by having the `Connection` passed in as a parameter. That is why the `getDistanceFrom(Connection, int)` needs a `Connection` object as a parameter.

Other methods in the `City` class access the database. For example, look at the following methods in the javadoc or in the source file:

- `findCity (Connection, String)`
- `findCity (Connection, int)`
- `findCity (Connection, String, String)`
- `getDistanceFrom(Connection, int)`
- `getTimeDifference(Connection, Date, String)`

These methods can be executed in a Java application as application-side JDBC methods or in the DBMS as database-side JDBC methods. In the section after this one, you will run a program that executes these methods. In the section after that, you will execute them from within `ij`.
Execute Methods in a Program as Application-Side Methods

CityTest2 is another program that should be in your working directory. It is identical to CityTest, with a couple of exceptions:

- It establishes a connection to the database and assigns it to a variable called conn.
- There is no city2 object (Paris) initialized in memory. Instead, the application uses the city_id for Paris, 35, or the string for Paris’s airport, “CDG”.

The method calls do not use the Paris City object, but instead use the JDBC methods that take the Connection and city_id code or the airport code and then access the database.

Quiz: Why do those methods take Connections as parameters?

Answer: Because they need a Connection to access the database.

1 Look at CityTest2.java. Note the missing Paris City object and the new method signatures.

2 Open a command window and change directories to the your_tutorial_home directory.

3 Run your setclasspath script.

4 Compile and run CityTest2:

   javac CityTest2.java

   java -Dcloudscape.system.home=your_tutorial_system CityTest2

Note that you need to set the cloudscape.system.home variable, because this program interacts with Cloudscape.

The output should be similar to that for CityTest.

Execute Methods in Statements as Database-Side Methods

Let’s take one of the methods executed in CityTest2:

getDistanceFrom(conn, 35)

and execute it in an SQL-J statement within ij.

In an SQL-J statement, you have no way of referring to an application variable for the connection. Instead, Cloudscape supplies a built-in function that gets the current
connection and passes it as a nested connection to the calling method. You call that function like this:

```
GETCURRENTCONNECTION()
```

Case does not matter.

In this section you will execute that method for all cities in the City database whose city_id is less than 10.

1. Start ij and get a connection to toursDB as outlined in “Start ij” on page 6-6 and “Connect to toursDB” on page 6-6. You should be connected to toursDB. For this lesson, you will turn off auto-commit. Auto-commit mode means that when a statement is completed, the method commit is called on that statement automatically. Committing a statement means that any changes it made in the database are final. If you never commit a statement, any changes it made in the database are only temporary. The concept of committing a statement or set of statements allows you to group statements together into atomic transactions. You can commit a transaction to make all changes permanent in the database. Or you can roll back a transaction to undo all the changes made by the transaction. Applications typically roll back a transaction if something goes wrong during one of the statements. Cloudscape also rolls back a transaction at runtime if a very serious database error occurs. You will learn more about committing and rolling back a transaction in Lesson 9, “Working with Connections and Transactions”. Auto-commit in effect makes every SQL-J statement a transaction, including basic queries. You will not want to automatically commit all the statements you execute in this chapter.

2. Turn off auto-commit using an ij command:

```
autocommit off;
```

3. Execute the following SQL-J statement, which finds the distance from Paris for those cities for which the city_id is less than 10:

```
SELECT city.getName(),
city.getDistanceFrom(GETCURRENTCONNECTION(), 35) AS DISTANCE_FROM_PARIS FROM CITIES
WHERE city_id < 10;
```

4. Find those European cities that are within 300 miles of Paris:
VALUES Expression

```
SELECT city.getName()
FROM Cities JOIN Countries
ON Cities.country_ISO_code = Countries.country_ISO_code
WHERE region = 'Europe' AND
city.getDistanceFrom(GETCURRENTCONNECTION(), 35) < 300;
```

5 Here’s another way to get the same results:

```
SELECT city.getName() FROM Cities JOIN Countries
ON Cities.country_ISO_code = Countries.country_ISO_code
WHERE region = 'Europe'
AND city.getDistanceFrom(
    (SELECT CITY FROM CITIES WHERE city_id = 35)) < 300;
```

6 This one shows the current time difference between Paris (city_id = 35) and New York (airport JFK):

```
SELECT city.getName(),
    city.getTimeDifference(getCurrentConnection(),
    CURRENT_DATE, 'JFK') AS CurrentTimeDifFromNewYork
FROM Cities WHERE city_id = 35;
```

7 Commit to release resources:

```
commit;
```

8 Leave your ij connection open for the next section.

VALUES Expression

Before you do the next section, you have to learn a little more about SQL and SQL-J.

As you saw in Chapter 4, “Objects and SQL”, SELECT is the SQL-J command that lets you retrieve data from a table. UPDATE and DELETE modify and delete data stored in a table. There is another basic SQL-J command that does not iterate over values in a database; it simply constructs a value or values. This command is VALUES.

Use the VALUES Command

The VALUES clause is used in an INSERT statement.

1 Execute the following statement in your open ij window:
Working with SQL-J’s Java Extensions

```
INSERT INTO COUNTRIES VALUES ('Bhutan', 'BH', 'Asia');
```

2 Check to see if the value is actually in the database:

```
SELECT * FROM COUNTRIES WHERE country_ISO_code = 'BH';
```

3 Don’t make this change in the database permanent, however; issue the ij rollback command:

```
rollback;
```

4 Check that Bhutan is no longer in the database:

```
SELECT * FROM COUNTRIES WHERE country_ISO_code = 'BH';
```

In an INSERT statement, the VALUES clause is used to specify which values get inserted.

5 However, the VALUES command can also stand on its own. Try executing the following statement:

```
VALUES ('Bhutan', 'BH', 'Asia');
```

It “returns” three String values, just as if you had selected them from some table. This VALUES expression essentially constructs a virtual table of one row and three columns.

The VALUES expression is useful in many cases. One example is that it allows you to execute system-supplied functions that return values.

6 Try this one:

```
-- returns the current date
VALUES CURRENT_DATE;
```

7 Leave ij running and your connection open.

Accessing Classes and Class Aliases

All the examples shown so far use methods associated with objects stored in the database.

You can invoke public methods and access public fields of any Java class that is visible to your JVM, provided that it is visible to the JVM in which Cloudscape is running. Instances of those classes do not have to be stored in the database.
For example, you can use a VALUES expression to construct a value, then execute a method on that value.

A class is visible when:

- It is public.
- It is installed on the local computer.
- It is in a library in the current class path.

When specifying a class name rather than referencing an instance of the class, you have to specify the full package name for the class unless you create a class alias for it. (Classes provided by Cloudscape come with built-in class aliases).

You will learn more about class aliases later on.

### Execute a Method in a VALUES Expression Using an Instance

Each built-in SQL-J type has a corresponding Java type. An SQL-J INT’s corresponding Java type is `java.lang.Integer`; an SQL-J CHAR’s corresponding Java type is `java.lang.String`. You can use a VALUES expression to construct an INT or CHAR value, then execute a method off its corresponding Java type.

For example:

- \(1+3\)
  
  Evaluates to an SQL-J INT, and is thus an instance of `java.lang.Integer`

- ‘Santiago’
  
  Evaluates to an SQL-J CHAR, and is thus an instance of `java.lang.String`.

1. Execute the following VALUES expressions in ij:

   ```
   VALUES((1+3).toString());
   VALUES('Santiago'.toUpperCase());
   ```

   `toString()` is a method in `java.lang.Integer`. `toUpperCase()` is a method in `java.lang.String`.

2. You can also invoke static methods associated with an instance of a Java class.

   For example, invoke the static method `toString(Integer)` of the `java.lang.Integer` class on the expression \(1+3\):

   ```
   VALUES((1+3).toString(35));
   ```

   The value returned (the string “35”) doesn’t have anything to do with the expression \(1+3\). Using an expression simply allowed you to create an
instance of the class `java.lang.Integer` from which to execute the static method.

**Execute a Method Using a Class, Not an Instance**

1. Finally, invoke a static method not associated with an instance of a Java class.

   In a Java program, you invoke a static method by specifying the class name. In an SQL-J statement, you may have to use the keyword `CLASS` before the class name to invoke a static method. You must also specify the full package and class name. (The names of java classes and packages are case-sensitive). Use the VALUES expression to call such methods when you want to retrieve its value.

   Execute the following statement:

   ```sql```
   VALUES ((CLASS java.lang.Integer).toString(35));
   ```

   **Quiz:** What data type is 35 in the above method call?

   **Answer:** This is a trick question. As you know, the literal 35 evaluates to a `java.lang.Integer`. However, there is no method with the signature `toString(java.lang.Integer)` in the `java.lang.Integer` class. If Cloudscape cannot find a method with a matching signature, it tries again, assuming that the literal maps to the corresponding primitive data type (in this case, an `int`). There is a method of the signature `toString(int)` in the `java.lang.Integer` class, and the call executes successfully.

**Work with Class Aliases**

You may have noticed that in the last query you invoked, you had to type the full package name for the class `java.lang.Integer` in addition to using the keyword `CLASS`. Cloudscape allows you to avoid both of those requirements if you define an alias for a class within your database.

Typically, the alias for a class name is just the class name (minus all the package names). This is similar to how the `import` statement works within a Java program. Class aliases are case-insensitive, however.

1. Create a class alias for `java.lang.Integer`.

   For this class, we can’t use the default alias, `Integer`, because `Integer` is a reserved word. (Cloudscape already uses it for a built-in data type). We’ll
have to specify an alternate alias name. Let’s use the same word, but delimit it with double quotes.

2 Execute the following SQL-J statement:

```
CREATE CLASS ALIAS "Integer" FOR java.lang.Integer;
```

You can now reference this class with the word "Integer" alone.

3 Try the query from the last task in its new, shortened form:

```
VALUES "Integer".toString(35);
```

### Execute Some More Methods Using Class Aliases

1 You can execute some methods off of the class `JBMSTours.serializabletypes.City`. Luckily, the class already has an alias for it, City. Execute the following SQL-J statement:

```
VALUES City.findCity(GETCURRENTCONNECTION(), 1);
```

In this case, the parameter to the VALUES expression actually does get data out of a table, so you would not expect it to be in a VALUES expression. However, the SQL-J statement that accesses the data is being called by the database-side JDBC method and is “hidden” from the outer statement.

**Quiz:** What statement would you have to use if the class did not have a class alias?

**Answer:**

```
VALUES ((CLASS JBMSTours.serializabletypes.City).findCity(
    GETCURRENTCONNECTION(), 1));
```

2 Execute an instance method off that last VALUES expression:

```
VALUES City.findCity(GETCURRENTCONNECTION(), 1).getName();
```

**NOTE:** To learn more about invoking static methods, which are the SQL-J constructs most like standard database stored procedures, see “Static Method Aliases” on page 6-18.
Use a Constructor

Cloudscape supports constructor methods. That means you can construct new instances of classes within an SQL-J statement, where applicable. If an alias is defined for the class, you can use the alias. Otherwise, specify the full package and class name.

1. Execute the following in ij:

```
VALUES (new java.lang.Integer('1'));
```

2. Here’s another one:

```
-- Location is an alias for
-- JBMSTours.serializabletypes.Location
VALUES new Location(52, 21, 'N', 4, 55, 'E');
```

(ij truncates the display.)

3. Here’s another:

```
-- Person is an alias for
-- JBMSTours.serializabletypes.Person
VALUES (new Person('Your', 'Name'));
```

4. Which leads to:

```
INSERT INTO people VALUES (26, new Person('Your', 'Name'));
```

5. Don’t make this change in the database permanent, however; issue the ij rollback command:

```
rollback;
```

Static Method Aliases

If you call some static method names often, you may get tired of specifying the full class name or even the class alias:

```
VALUES (CLASS java.lang.Math).sqrt(25.0);
```

Cloudscape allows you to create aliases or shorthand names for static methods. These aliases are database-specific. If you want to use them in more than one database, you must define them in each database.
Create a Method Alias

To create a method alias, you use the CREATE METHOD ALIAS command.

1. Execute the following statement in ij:

   CREATE METHOD ALIAS sqrt FOR java.lang.Math.sqrt;

   Once you create the alias, you can execute this static function without having to type the full package and class name, and without having to use the CLASS keyword.

2. Execute the following statement:

   VALUES (sqrt(25.0));

3. Execute this statement:

   CREATE METHOD ALIAS abs FOR java.lang.Math.abs;

4. Execute the following statement:

   VALUES (abs(-3));

   toursDB has some method aliases already defined for you.
   One alias that returns a value is called findcity, and it’s an alias for JBMSTours.serializabletypes.City.findCity.

5. Before executing it, let’s expand the maximum display width of columns in ij:

   maximumDisplayWidth 100;

Execute Method Aliases for Database-Side Methods

When static methods are database-side JDBC methods, they resemble traditional stored procedures. Creating method aliases for these methods makes them easier to use.

1. To execute a static method that returns a value or a method alias for such a method, you use the VALUES expression. Execute these:

   VALUES (findcity(GETCURRENTCONNECTION(), 1).toString());

   VALUES (findcity(GETCURRENTCONNECTION(), 'Amsterdam', 'Netherlands').toString());

   VALUES (findcity(GETCURRENTCONNECTION(), 'AMS').toString());

   Note that the alias finds all signatures of the method.
To execute a static method that does not return a value or a method alias for such a method, you use the CALL statement. The method alias cleanOutFlightAvailability is an alias for the static method JBMSTours.FlightBuilder.cleanOutFlightAvailability. That method deletes all entries in the FlightAvailability table that are earlier than the current date; customers do not need to know the availability of flights that have already occurred. Try executing it:

```sql
CALL cleanOutFlightAvailability(GETCURRENTCONNECTION());
```

The product installation includes scripts for creating method aliases for the static methods of java.lang.Math. See the Cloudscape Developer’s Guide for information.

3 In ij, disconnect from toursDB:

```sql
disconnect;
```

4 Shut down the system and exit using the exit command:

```sql
exit;
```
7 Programming for Performance

Concepts covered:

- The performance advantages of prepared statements
  “Prepared Statements” on page 7-2
- Avoiding compilation time with stored prepared statements
  “Stored PreparedStatements” on page 7-8
- Getting the best performance out of queries
  “Performance, Optimization, and Indexes” on page 7-12

Tasks:

- “Prepare a Statement in ij” on page 7-4
- “Execute a Prepared Statement in ij” on page 7-5
- “Compare the Insert Performance of PreparedStatements to that of Statements” on page 7-6
- “Create a Stored Prepared Statement” on page 7-8
- “Work with One of the Stored Prepared Statements in toursDB” on page 7-9
- “Execute a Method That Uses Stored Prepared Statements” on page 7-11
- “Compare the Performance of Statements, PreparedStatements, and Stored Prepared Statements” on page 7-11
- “Find the Name of the Index on the orig_airport Column” on page 7-14
- “Execute a Query with and without an Index” on page 7-15
- “Execute a Query with a More Complex Comparison” on page 7-15
- “Execute the Query That Uses the Column in an Expression” on page 7-16
- “Execute a Query in Which the Index Covers the Query” on page 7-17
Prepared Statements

A prepared statement is an SQL-J statement that is compiled once for a particular Connection and is then available to that Connection to be executed many times until explicitly or implicitly closed.

Using JDBC, you create a PreparedStatement, which you explicitly close with the close method, or implicitly close by disconnecting. It is also implicitly closed when garbage-collected.

Compilation is a time-intensive process composed of four stages:

- **Parsing**
  The stage in which the SQL-J language parser reads the text of the statement, determines which words are commands, which are identifiers, and which are valid expressions, checks for valid syntax, and generates an internal tree representation of the statement.

- **Binding**
  The stage in which Cloudscape verifies the names of columns, tables, and other database objects found in the statement, determines the data types of all expressions in the statement, and notes dependencies.

- **Optimization**
  The stage in which Cloudscape chooses the access path for statement execution (whether to use an index, which tables to scan, etc.), join order, locking granularity, and notes dependencies on indexes.

- **Code generation**
  The final stage in compilation, in which Cloudscape generates the actual code to execute.

Execution is the actual evaluation of the statement.

For example, consider the following SQL-J statement:

```sql
SELECT rooms_taken
FROM HotelAvailability
WHERE hotel_id = 10
AND booking_date = DATE’1998-06-09’
```

In the compilation stage, Cloudscape takes the following actions:
• parses the text of the string, finding the keywords SELECT, FROM, WHERE, and AND, along with identifiers and literals

• verifies the names of the columns and the tables, checks data types and verifies that the data type of hotel_id is the same as that of the literal 10 and that the data type of booking_date is the same as the literal Date’1998-06-09’, and notes that the statement depends on the HotelAvailability table (you won’t be able to delete that table until you’re finished with the statement)

• determines that it can make use of the primary key backing index on the hotel_id and booking_date columns

• generates code that it will use to evaluate the statement

In the execution stage, Cloudscape actually evaluates the statement; it retrieves records from the HotelAvailability table that match the search condition (using the index to shorten retrieval time), filters out all columns except rooms_taken, and then returns results.

If an application executes the same statement more than once, it will achieve substantial performance improvements by using a prepared statement instead of a statement, because it avoids repeat compilation time.

Often an application executes statements that are similar (but not exactly alike) more than once. For example, the JBMSTours application executes a statement similar to the one shown above every time it checks on the availability of a particular hotel on a particular date. However, the application does not know in advance that the actual hotel_id to look for will be 10 or that the date will be Date’1998-06-09’. Instead, it prepares a single PreparedStatement once that uses the dynamic or IN parameter construct of the PreparedStatement interface. Instead of using the literals 10 or Date’1998-06-09’, the application uses ?s (placeholders) for these two parameters.

Each time it executes the statement, it provides the actual values of these variables.

Using parameters, the application could prepare the following statement:

```sql
SELECT rooms_taken
FROM HotelAvailability
WHERE hotel_id = ? AND booking_date = ?
```

To look up the number of rooms taken for hotel number 5 on January 9, 1998, the application fills in the particular parameters using JDBC methods, not SQL-J, and then executes the precompiled statement.
PreparedStatement ps = conn.prepareStatement('SELECT rooms_taken FROM HotelAvailability' + 'WHERE hotel_id = ? AND booking_date = ?');
ps.setInt(1, 5);
ps.setDate(2, new java.sql.Date(98, 1, 9));
ResultSet rs = ps.executeQuery();

Dynamic parameters (?) cannot be used indiscriminately within PreparedStatements; Cloudscape has to be able to infer data type information about each parameter. In the above statement, Cloudscape infers that the first parameter is an INTEGER data type (the type of hotel_id) and that the second parameter is a DATE data type (the type of booking_date). Cloudscape expects the application to provide the JDBC data types that correspond to those types and to use the appropriate setXXX methods for those types. (You will learn about those methods in a later task.) At execution time, supplying parameters of other data types may cause an error. (Cloudscape handles some conversions automatically, such as to and from string data types and between numerical data types.)

**Quiz:** Can Cloudscape infer the data type of the dynamic parameter in the following SQL-J statements?

SELECT ? FROM HotelAvailability
VALUES (CLASS myClass).overloadedMethod(?)

**Answer:** No. The HotelAvailability table has many columns of differing data types. The ? could stand for any of those columns, so this statement is not allowed. In the second example, the method is overloaded, and Cloudscape cannot guess the data type of the parameter. You can get around this second limitation by using the CAST command to cast its data type:

VALUES (CLASS myClass).overloadedMethod(CAST (?) AS DECIMAL)

**Prepare a Statement in ij**

In this task, you will use ij to prepare and execute prepared statements.

1. Start ij and connect to toursDB following the instructions in “Start ij” on page 6-6 and “Connect to toursDB” on page 6-6.

2. Turn off auto-commit.

   autocommit off;

   Ij allows you to create a prepared statement using the following syntax:
PREPARE Identifier AS String;

where Identifier is your name for the prepared statement and String is the text of the statement. The prepared statement is known by this name only within ij.

3 Prepare the statement in the above example, calling it findRoomsTaken:

PREPARE findRoomsTaken AS
  ‘SELECT rooms_taken
   FROM HotelAvailability
   WHERE hotel_id = ?
   AND booking_date = ?’;

**Execute a Prepared Statement in ij**

After you prepare a statement in ij, you execute it by using ij’s Execute command and the name you gave it. To supply the values of dynamic parameters, you provide a result set within a string with the expected number of columns—the results of a query—using an SQL-J statement string. The statement will be executed once for each row in the result set.

In the case of findRoomsTaken, you would need to supply a result set with two columns: the first of the INTEGER data type, the second of the DATE data type. The easiest way to do this is to use a VALUES clause.

For example, to provide the parameters 10 and today’s date, you could use:

VALUES (10, current_date)

1 Execute the command using the parameters 10 and today’s date:

execute findRoomsTaken using ‘VALUES (10, current_date)’;

Note: If no rooms have been booked for that hotel on that particular date, the query won’t return anything.

2 Execute the command using the parameters 50 and a date of your choice:

execute findRoomsTaken using ‘VALUES (50, DATE’’1998-08-15’’)’;

(The date literal inside a string requires doubled single quotes.)

Alter the values you supply until you actually get some results.

You can also supply the values from a query. We could supply values to findRoomsTaken using a simple query from the HotelBookings table.
Programming for Performance

```sql
SELECT hotel_id, arrival FROM HotelBookings

Depending on how many times you ran JBMTours.BuildATour, this table may contain more than one row.

3 Execute findRoomsTaken using a SELECT statement:

execute findRoomsTaken using
  'SELECT hotel_id, arrival FROM HotelBookings';

<table>
<thead>
<tr>
<th>Quiz: Does the above ij statement avoid all compilation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answer: No. Cloudscape has to compile the new SELECT statement used to provide the parameter values.</td>
</tr>
</tbody>
</table>

4 Disconnect:

disconnect;

5 Shut down the system and exit ij with the exit command:

exit;

Such a construct does not actually save you much compilation time, because Cloudscape has to compile the SELECT statement you used to provide the result set.

A Java programmer does not have the same limitation. In a Java program, you use the methods of java.sql.PreparedStatement to provide the parameter values, which do not take any Cloudscape compilation time.

In the next section, you will work with a Java program.

Compare the Insert Performance of PreparedStatements to that of Statements

Prepared statements are a good choice for INSERT statements, which are typically executed several times.

In this task, you will use a Java program to insert data into the HotelBookings table using a Statement and then a PreparedStatement.

1 Open the Java file InsertComparer.java, which you should have copied into Learning Cloudscape
your working directory from the scripts directory.

This simple program connects to toursDB, inserts 16 rows into the HotelBookings table using 16 different statements, and then prints the time it took to accomplish the inserts.

Then it prepares a single prepared statement, executes it 16 different times with different parameters, and then prints the time it took to accomplish the inserts.

2 Look at the first INSERT statement.

```java
s.executeUpdate("INSERT INTO HotelBookings VALUES (250, 10, DATE '1997-10-15', DATE '1997-10-20', 3, 1000.00)");
```

When Cloudscape compiles the statement, it must determine the data type of all six values and compare them to the data types of the six columns in the table. That is a lot of work.

3 Look at the PreparedStatement and the first execution of it:

```java
PreparedStatement ps = conn.prepareStatement(
"INSERT INTO HotelBookings VALUES (?, ?, ?, ?, ?, ?)");
ps.setInt(1, 250);
ps.setInt(2, 10);
ps.setDate(3, new java.sql.Date(98, 11, 15));
ps.setDate(4, new java.sql.Date(98, 11, 20));
ps.setInt(5, 3);
ps.setObject(6, new java.math.BigDecimal(1000.00));
ps.executeUpdate();
```

When Cloudscape compiles the PreparedStatement, it determines what data type each of the dynamic parameters must be.

Each time the application executes the statement, it provides values using the setXXX methods of the java.sql.PreparedStatement interface.

There is one set method for each standard data type in java.sql, plus one for Java objects. These methods take two parameters: the first is the number of the parameter, and the second is the actual value to supply. The parameters are ordered from left to right, starting with 1. The application takes responsibility for providing a value of the expected type with the expected setXXX method.

4 Compile the program:

```bash
javac InsertComparer.java
```

(You may get a note about deprecated APIs, depending on the version of the JDK that you are using. You can ignore these messages.)
5 Run the program:

```java
java -Dcloudscape.system.home=your_tutorial_system
InsertComparer
```

Note the time it took to execute. You should get output looking something like this:

```
Compiling and executing 16 statements took 4917 milliseconds.
Compiling the statement once and executing it 16 times took 601 milliseconds.
```

The actual time it took will vary, depending on your system. In our environment, using a `PreparedStatement` provided a substantial performance gain.

## Stored PreparedStatements

Within Cloudscape, prepared statements exist only within the context of a single user’s session, and they disappear when garbage-collected from the user’s application. You must re-create them for every session.

Cloudscape allows you to store and name precompiled statements for use across multiple sessions. Stored prepared statements are usually compiled only once (when you create them), not for every session. You create and name stored prepared statements with a language command. In the previous section, you named a prepared statement, but that name existed only within the context of `ij`. Stored prepared statements have names that are durable across sessions, like tables, views, and other database objects.

Since stored prepared statements usually need to be compiled only once, they are a great performance improvement for applications that use a lot of prepared statements or very complex statements or for applications that have fixed statements and are run repeatedly.

### Create a Stored Prepared Statement

You can create a stored prepared statement with (you guessed it) a `CREATE STATEMENT` statement. Cloudview provides a GUI environment that makes it even easier to work with them.

In this task you will create a stored prepared statement to match the `un`stored prepared statement you created earlier in “Prepare a Statement in `ij`” on page 7-4.
1. Start Cloudview as described in “Start Cloudview in the Home Directory” on page 3-3. Then open a connection to *toursDB* as described in “Open a Connection to *toursDB*” on page 3-9. Select the *toursDB* icon in the left-hand hierarchy window. Click the “+” to the left of the icon.

2. Click the Stored Statements icon.

3. Choose New->Stored Statement from the Edit menu.

4. Type *findRoomsTaken* in the Name box.

5. Type the following SQL-J statement in the Query box:

   ```sql
   SELECT rooms_taken FROM HotelAvailability WHERE hotel_id = ?
   AND booking_date = ?
   ```

6. Click OK to create the statement.
   
   You can execute the statement in the same window.
   
   The statement has two parameters (you know this because the statement contains two question marks).

7. Fill in the parameter window, like this (substitute the current date for the date shown):

   ```
   9, 1999-09-01
   ```

8. Click the *toursDB* icon to get to the SQL window.

9. Now execute the statement again from the less-friendly SQL-J environment.

   You will have to provide the parameters with a VALUES statement:

   ```sql
   EXECUTE STATEMENT findRoomsTaken USING VALUES (9, current_date);
   ```

10. Keep the connection open for the next task.

**Work with One of the Stored Prepared Statements in *toursDB***

The *toursDB* database has four stored prepared statements, which are used by the *FlightBuilder* class to query the *Flights* table and to insert into the *FlightObjects* table. It is especially useful to be able to store and precompile the complex statement that looks for transfer flights between two cities. Let’s look at the SQL-J for that statement (it’s a doozy):

```sql
SELECT firstleg.flight_id, firstleg.segment_number, secondleg.flight_id, secondleg.segment_number
FROM (SELECT flight_id, segment_number, arrive_time,
```
The statement is a good example of the power of SQL-92 syntax (it uses derived tables, which you will learn about in the next chapter). It is so complex that it can take a few seconds to compile. However, in our database, this is a stored prepared statement, so our application can avoid that compilation time.

The application creates and stores the stored prepared statements in the class `JBMSTours.CreateSchema`. This one has the name `getFullTransferFlightInOneBlow` (because it gets both legs of the flight in one statement).

1. Execute the `getFullTransferFlightInOneBlow` statement, using the values ‘SFO’ and ‘GRU’ (the airport codes for San Francisco and São Paulo):
2. Select the Stored Statements icon, and click the plus sign to the left of it to display statements if necessary.
3. Select GETFULLTRANSFERFLIGHTINONEBLOW.
4. Fill in the parameters like this:
   ```
   SFO, GRU
   ```
   Do not enclose the strings in single quotes. Don’t forget the comma separate the strings.

   The statement returns rows that consist of the flight ids and segment numbers of two different flights that will be combined to make one transfer flight.

   In Cloudview, you provide parameters with the Parameters box. In ij or an SQL-J Statement, you provide parameters with an SQL-J statement.

   In a Java application, you can set the parameters using the `setXXX` methods of `java.sql.PreparedStatement` and avoid all compilation costs. (You will learn about an example class that does this in “Compare the Performance of Statements, PreparedStatements, and Stored Prepared Statements” on page 7-11.)
Execute a Method That Uses Stored Prepared Statements

You can make any statement a stored prepared statement. The JBMSTours.FlightBuilder class has one static method that uses stored prepared statements, so it should run very quickly. You can execute this method in Cloudview. This class has an alias.

1. Click the toursDB icon to return to the SQL window. Execute this statement:

   ```java
   VALUES FlightBuilder{returnAnyFlight(
      getCurrentConnection(), 'SFO', 'MAD').toString();
   
   The method returns a String version of a Flight object, not just raw data.

2. Execute the `returnAnyFlight` static method to find out if there are any flights (direct or not) between New York and Cairo:

   ```java
   VALUES FlightBuilder{returnAnyFlight(
      getCurrentConnection(), 'JFK', 'CAI').toString();
   
   Look at the source code for the FlightBuilder class to see which stored prepared statements it uses and to see how it sets parameters.

3. Close the connection and exit Cloudview.

Compare the Performance of Statements, PreparedStatements, and Stored Prepared Statements

Remember InsertComparer? There’s a similar class called SelectComparer that compares the performance of using Statements, PreparedStatements, and stored prepared statements.

1. Open the Java file SelectComparer.java if you wish to examine it. You should have copied into your working directory from the scripts directory.

2. Compile the program:

   ```bash
   javac SelectComparer.java
   
3. Run the program:

   ```bash
   java -Dcloudscape.system.home=your_tutorial_system
   SelectComparer
   
   Note the time it took to execute. You should get output looking something like this:

   ```none
   Compiling and executing 16 statements took 5749 milliseconds.
Compiling the statement once and executing it 16 times took 751 milliseconds.

Retrieving a stored prepared statement without compiling it and executing it 16 times took 641 milliseconds.

The actual time it took will vary, depending on your system. In our environment, using a `PreparedStatement` provided a substantial performance gain over using a series of `Statements`, and using a stored prepared statement provided a small performance gain.

**Performance, Optimization, and Indexes**

In a Cloudscape system, units of storage are called *conglomerates*; they are either tables or indexes.

An index is a database object that can improve the performance of some statements. You create an index on a column (or columns) in a table to provide a fast lookup of the values stored in that column. In a table, rows are stored in the order in which they are inserted—that is, in no useful order. Deletes, updates, and new inserts can rearrange this order. Tables are sometimes called heaps for this reason. In indexes, Cloudscape stores the values of the indexed column or columns in order in a tree structure called a BTREE. Depending on the amount of data, Cloudscape provides a number of nodes in the tree to provide quick lookup of values.

Figure 7-1 shows the “top” of the `Flights` table, where the rows appear in no particular order (well, in the order in which they were inserted).

![Figure 7-1](image)

Partial view of the `Flights` table
The `orig_airport` column is indexed. Its index stores every value in the `orig_airport` column plus information on how to retrieve the entire corresponding row for each value, as shown in Figure 7-2:

- For every row in `Flights`, there is an entry in the index that includes the value of the `orig_airport` column and the address of the row itself. The entries are stored in ascending order by the `orig_airport` values. This level is called the leaf level of the index.
- One or more abstract levels in the BTREE structure have values that point into lower levels of the index, much like tab dividers in a three-ring notebook that help you find the correct section quickly. These levels help Cloudscape determine where to begin an index scan.

![Figure 7-2](image)

**Figure 7-2** The index on the `orig_airport` column helps Cloudscape find the rows for which `orig_airport` is equal to a specific value.

### Types of Queries for Which Indexes Are Useful

Queries that are aided by indexes usually have WHERE clauses. A WHERE clause means that you want to restrict the results returned to specific records. Retrieving only some rows from a table is called restriction. A query without a WHERE clause usually means you want all the records in the table, even if you want only specific columns. Retrieving only some of the columns in a row is called projection.

Indexes are useful in a few cases:

- restrictions (discussed in this section)

  If you want all the rows in the table, it doesn’t make sense to find the locations of specific rows; you want all of them.
Restricting statements that can make use of an index provide starting and stopping points for an index scan in a WHERE clause. That is, they tell Cloudscape the point to begin its scan of the index and where to end the scan.

The simplest example of such a statement is one in which the WHERE clause specifies a specific value for the indexed column, in which case the stop and start positions are the same. For example:

`SELECT * FROM Flights WHERE orig_airport = 'SFO'`

Cloudscape can use the lookup nodes in the BTREE index to get to the value SFO very quickly.

More complex comparisons in the WHERE clause also provide start and stop positions. An example is a statement with a WHERE clause looking for rows for which the `orig_airport` value is less than BBB. This means that Cloudscape must begin the scan at the beginning of the index; it can end the scan at BBB.

An index scan that uses start and stop conditions is called a matching index scan.

As it turns out, the `Flights` table does indeed have an index on the `orig_airport` column. In the next few tasks, you will compare the performance of retrieving a subset of rows from the table with and without the index.

### Find the Name of the Index on the `orig_airport` Column

First, let’s find out the name of the index on the `orig_airport` column. The easiest way to do that is with Cloudview.

1. Start Cloudview as described in “Start Cloudview in the Home Directory” on page 3-3. Then open a connection to toursDB as described in “Open a Connection to toursDB” on page 3-9.
2. Select the `toursDB` icon in the left-hand hierarchy window. Click the “+” to the left of the icon to display the Table icon.
3. Click the “+” to the left of the `Flights` table to display its indexes.
4. Click the Indexes beneath the `Flights` table icon.

Indexes appear in the Name box (on the right). (Indexes that are created to enforce or back up primary, unique, and foreign key constraints are not displayed here.)
5 ORIGINDEX appears in the Name box; double-click it. It is defined on the orig_airport column.

**Execute a Query with and without an Index**

1 Click the toursDB icon in the left-hand hierarchy window to display the SQL window.

2 Execute the following query from the SQL window:

   ```sql
   SELECT * FROM Flights WHERE orig_airport = 'SFO'
   ```

   The execute time should appear in the bottom bar below the results; write down this number.

3 Now delete the index:

   ```sql
   DROP INDEX OrigIndex;
   ```

4 Now re-execute the query (use the back arrow in the SQL window to return to the query, then click Execute).

5 Note the new execute time.

   In our system, the query execution time using the index was 0.251 seconds, and without the index the execution time was 0.922 second.

6 Re-create the index:

   ```sql
   CREATE INDEX OrigIndex ON Flights(orig_airport);
   ```

   Re-creating the index takes longer than the query, because Cloudscape has to rebuild the BTREE structure and order the data.

   There is an easier way to prevent Cloudscape from using a useful index: using a PROPERTIES clause to specify a null value for index. When the index value is set to null, Cloudscape must do a full table scan. In the following sections, you will use a PROPERTIES clause to test execution time without an index.

**Execute a Query with a More Complex Comparison**

1 Execute the following query from the SQL window:

   ```sql
   SELECT * FROM Flights
   WHERE orig_airport BETWEEN 'SCL' AND 'SFO'
   ```

   The execute time should appear in the bottom bar below the results; write down this number.
Now execute the query with a PROPERTIES clause that sets the index value to null:

```sql
SELECT * FROM Flights PROPERTIES index=null
WHERE orig_airport BETWEEN 'SCL' AND 'SFO'
```

Note the new execute time.
In our system, the query execution time using the index was 0.15 seconds, and without the index the execution time was 1.192 seconds.

**Execute the Query That Uses the Column in an Expression**

In the previous examples, the reference to the indexed column was “simple”; it was not used within an expression. Here are two examples of a reference to a column used within an expression:

- `orig_airport.toLowerCase()`
- `orig_airport || 'hello, world'`

|| is the concatenation operator; it concatenates two strings, or appends one string to another.

1. Execute the following query from the SQL window:

```sql
SELECT * FROM Flights
WHERE orig_airport.toLowerCase() = 'sfo'
```

The execute time should appear in the bottom bar below the results; write down this number.

2. Now re-execute the query, forcing the table scan:

```sql
SELECT * FROM Flights PROPERTIES index=null
WHERE orig_airport.toLowerCase() = 'sfo'
```

3. Note the new execute time.
In our system, the query execution time using the index was 0.06 seconds, and without the index the execution time was 0.19 seconds.
In this example, there is no substantial difference in execution time. That’s because Cloudscape cannot use an index if the WHERE clause uses the column name within an expression.
Covering Indexes

Even when there is no definite starting or stopping point for an index scan, an index may speed up the execution of a query if the index covers the query. An index is said to cover the query if all the columns specified in the query are included in the index. These are the columns that are all columns referenced in the query, not just columns in a WHERE clause. If so, Cloudscape never has to go to the data pages but can retrieve all data from index access alone.

Execute a Query in Which the Index Covers the Query

1. Execute the following query and write down the execution time.

   SELECT orig_airport FROM Flights

Delete the Index and Re-Execute the Query

1. Now delete the index:

   DROP INDEX OrigIndex;

2. Now re-execute the query (use the back arrow in the SQL window to return to the query, then click Execute).

3. Note the new execute time.

   In our system, the query execution time using the covering index was about twice as fast as the query execution time not using the index.

4. Re-create the index:

   CREATE INDEX OrigIndex on Flights(orig_airport);

   Cloudscape can get all required data out of the index more quickly than from the table, since it has fewer pages to read.

RunTimeStatistics

Cloudscape provides a quick way to understand how a query was executed. If you turn on RunTimeStatistics and statistics timing attributes before executing a query, Cloudscape creates an object showing (among other things) access path and join order.

Access path means whether Cloudscape used an index or went directly to the table; join order is the order in which it examined the tables in a join.
Cloudview provides an easy-to-use interface for working with runtime statistics.

**View the RunTimeStatistics for a Query**

1. Select Use Statistics in the SQL window.
2. Execute the following query:

   ```sql
   SELECT Country, City
   FROM Countries, Cities
   WHERE Countries.country_ISO_code = Cities.country_ISO_code
   ```
3. Click the Statistics tab.

   ![Statistics Tab]

   Cloudview displays a tree-shaped structure representing the way Cloudscape accessed the table. This structure shows that Cloudscape accessed the `Cities` table first (table scan), then used the index on `Countries` to access the corresponding rows in that table.

   **NOTE:** The plan chosen by the optimizer may differ from the plan shown in the picture.

4. Exit Cloudview.

   **NOTE:** You will find more information on how to work with RunTimeStatistics in *Tuning Cloudscape*.

*Learning Cloudscape*
8 Virtual Tables, External Data, and Aggregates

Concepts covered:
- “Virtual (Derived) Tables and Views” on page 8-2
- “External Virtual Tables and Bulk Import” on page 8-5
- “Aggregate Data” on page 8-11
- “GROUP BY and GenerateReport” on page 8-13

Tasks:
- “Work with a Virtual Table” on page 8-2
- “Work with a View” on page 8-3
- “Import Data from a Flat File Using the Built-In Utility” on page 8-6
- “Import Data from a Flat File Using an SQL-J Statement” on page 8-7
- “Configure the ODBC Data Source for the Access Database” on page 8-9
- “Load the ODBC-JDBC Bridge Driver and Import Data” on page 8-9
- “Run JBMSTours.BuildATour a Few More Times” on page 8-11
- “Start ij and Work with Aggregates” on page 8-11
- “Work with the User-Defined Aggregate MAXBUTONE” on page 8-13
- “Use GROUP BY” on page 8-13
- “Use HAVING” on page 8-14
- “Run JBMSTours.GenerateReport” on page 8-15
Virtual (Derived) Tables and Views

Cloudscape’s flexible SQL-92 grammar allows you to construct virtual or derived tables on the fly within an SQL-J statement. You learned about the SQL-92 VALUES clause in Lesson 6, “Working with SQL-J’s Java Extensions”; the VALUES clause allows you to construct a virtual table that contains columns and rows that you can treat like any other table. Another way to think about it is as a constructed result set that can be referred to within an SQL-J statement.

Work with a Virtual Table

1. Start Cloudview as described in “Start Cloudview in the Home Directory” on page 3-3. Then open a connection to toursDB as described in “Open a Connection to toursDB” on page 3-9.
2. Then select the toursDB icon in the left-hand hierarchy window to display the SQL window.
3. Execute the following SQL-J statement:

```
VALUES ('orange', 'orange'), ('apple', 'red'),
('banana', 'yellow');
```

The results it returns look like a result set retrieved from a table; it returns three rows of two columns.

SQL-J allows you to treat such a virtual table as a real table within an SQL-J statement; you can SELECT an item from it, or use it in a UNION or JOIN operation. When a virtual table is part of a SELECT statement, it is enclosed within parentheses. In such a context, it is called a table expression. You will create such table expressions in the tasks in this section.

You typically must name your virtual table to perform such operations. This name is not permanent; it exists only within the scope of the statement. You name the virtual table using the AS keyword to give it a correlation name; column names are placed inside parentheses after the correlation name.
4 Name your virtual table *Fruits_And_Colors*; name the first column *fruit* and the second column *color*. Select all columns (*) from this table.

```sql
SELECT *
FROM
VALUES ('orange', 'orange'), ('apple', 'red'),
('banana', 'yellow')
AS Fruits_And_Colors(fruit, color);
```

5 Create another virtual table called *Fruits_And_Foods*, with two columns, *fruit* and *food*. Select all columns (*) from this table:

```sql
SELECT *
FROM
VALUES ('orange', 'orange juice'), ('apple', 'pie'),
('apple', 'muffins'), ('banana', 'bread'),
('banana', 'meringue pie')
AS Fruits_And_Foods(fruit, food);
```

6 Now join the two virtual tables:

```sql
SELECT color, food
FROM
VALUES ('orange', 'orange'), ('apple', 'red'),
('banana', 'yellow')
AS Fruits_And_Colors(fruit, color)
JOIN
VALUES ('orange', 'orange juice'), ('apple', 'pie'),
('apple', 'muffins'), ('banana', 'bread'),
('banana', 'meringue pie')
AS Fruits_And_Foods(fruit, food)
USING (fruit)
```

**Work with a View**

A view is a permanent virtual table that you create by naming an SQL-J statement. The *toursDB* schema contains one view. Once you create a view, it is always available within the database until you drop it, so it is similar to a stored prepared statement. However, there are some differences between views and stored prepared statements, as shown in Table 1 on page 8-4:
Virtual Tables, External Data, and Aggregates

Table 1 Differences Between Views and Stored Prepared Statements

<table>
<thead>
<tr>
<th>Feature</th>
<th>Views</th>
<th>Stored Prepared Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are named by user</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Are precompiled</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Contain dynamic parameters</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Are virtual tables and can take the place of a table in any SQL-J statement</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Since views create virtual tables, views can be part of SELECT statements. Views help you simplify or virtualize a database schema. For example, if your application interacts with views instead of with the underlying tables, you can make changes to the underlying tables and then alter the views to shield the application from the underlying changes.

1. Click the “+” next to the View icon in the left-hand hierarchy window to display the views within toursDB.
2. Select the view Segments_Seatbookings.
3. Select the View tab.
   - The Query window displays the text of the statement used to create the view. In a view statement, unlike in a table expression, you state the name of the view and its columns at the beginning of the statement. The text of the actual statement follows the AS keyword.
   - The view offers a simple view of the FlightBookings table.
   - The FlightBookings table is a bit complex, because it consists of one row for each flight booking. In the JBMSTours application, a Flight can consist of one or two flight segments. The FlightBookings table has two extra columns to store the flight_id and segment_number of the second segment if a group’s flight consists of two segments. If not, these two columns are null.
   - The organization of this table works well because it presents each flight, whether one or two parts, as a unit. For example, you can tell at a glance how many flights were booked.
   - However, what if you wanted to gather data about individual segments? The Segments_Seatbookings view presents the data as if the segments were booked individually. (You will use this view in the next section when you work with aggregates.)
4. Click the Execute button.
Clicking the Execute button is equivalent to executing the view’s SQL-J statement:

```java
SELECT * FROM Segments_SeatBookings
```

When you execute this statement, Cloudscape compiles the view’s underlying statement and then executes it. You can execute any SQL-J statement against the view.

Select the toursDB icon to get back to the SQL window. Then execute the following statement:

```java
SELECT Airline_Full, number_seats
FROM Airlines, Segments_SeatBookings
WHERE Airline = flight_id.substring(0,2)
```

**External Virtual Tables and Bulk Import**

Cloudscape provides a construct for presenting external data—from a flat file, from another vendor’s database, from a news feed—as a virtual table to an SQL-J SELECT statement. External data presented in this way constitutes an external virtual table that can be used within the scope of an SQL-J statement in the same way as other virtual or derived tables.

An external virtual table is created by an instance of any class that fulfills the requirements of Cloudscape’s virtual table interface (VTI):

- It implements the `java.sql.ResultSet` interface. It is up to the programmer to determine what work needs to be done in order to return the `ResultSetMetaData`.
- The parameters to the VTI constructor must all be constants to ensure that the result set description can be determined during compilation.

Optionally, the class:

- provides the static method `public static java.sql.ResultSetMetaData getResultSetMetaData(parameterList)`
  for which the `parameterList` signature is the same as for the constructor for that object that appears in the SQL-J statement.
- implements `COM.cloudscape.vti.VTICosting`

Cloudscape provides a class that fulfills these requirements, `COM.cloudscape.tools.FileImport`. This class can present data from a flat text file.
Virtual Tables, External Data, and Aggregates

You can write any class that fulfills the above requirements to “build your own” external virtual table. Later in this chapter, you will look at another class that fulfills those requirements to provide a gateway to any vendor’s database that has a JDBC driver.

Import Data from a Flat File Using the Built-In Utility

1. Close the connection to the **toursDB** database from Cloudview.
2. Open a connection to the **HelloWorldDB** database as described in “Connect to HelloWorldDB” on page 3-4.
3. Create a new table called **WorldCupStatistics** with the following columns:
   - RANK, SMALLINT
   - TEAM, VARCHAR(20)
   - P, SMALLINT
   - W, SMALLINT
   - D, SMALLINT
   - L, SMALLINT,
   - PTS, SMALLINT
   - PERCENT, DECIMAL(3,1)

   You can use Cloudview to create the table, following the instructions for table creation in “Create the Sayings Table” on page 3-4. Or, if you like, you can just cut and paste the following SQL-J statement into the SQL window:

   ```sql
   CREATE TABLE WorldCupStatistics
   (Rank SMALLINT,
    Team VARCHAR(20),
    P SMALLINT,
    W SMALLINT,
    D SMALLINT,
    L SMALLINT,
    Pts SMALLINT,
    Percent DECIMAL(3,1));
   ```

4. After you create the table, select its icon.
5. Choose Edit->Import.
6. Navigate to the **your_tutorial_home** directory. Select **wc_stat.dat**.
7. Click Open. If you are working on a PC platform, skip to step 9.
If you are working on a non-PC platform only, select the Advanced tab and select CR-LF as the record separator. (For users on PC platforms, CR-LF is assumed to be the default record separator.) Click Save to save the control file, which specifies non-default settings, for the import. Type `mycontro.ctl` as the name of your control file. Click Save. The file should be saved into the system directory.

Click OK.
Data is imported into the file.

Examine the data by clicking the Data tab.

**Import Data from a Flat File Using an SQL-J Statement**

1. Delete all the data from the `WorldCupStatistics` table using the following statement in the SQL window:

   ```sql
   DELETE
   FROM WorldCupStatistics
   ```

2. Click the lightning bolt to execute.

3. Import data from the same file with one of the following SQL-J statements, substituting the path to the `your_tutorial_home` directory where indicated.
   Users on PC platforms should use this statement:

   ```sql
   INSERT INTO WorldCupStatistics
   SELECT * from NEW FileImport('YOURTUTORIALHOME/wc_stat.dat') AS myExternalData;
   ```

   Users on non-PC platforms should use this statement (which specifies a control file):

   ```sql
   INSERT INTO WorldCupStatistics
   SELECT * from NEW COM.cloudscape.tools.FileImport('YOURTUTORIALHOME/wc_stat.dat',
   'mycontrol.ctl') AS myExternalData;
   ```

   **NOTE:** Do not split the string containing the file URL into two lines. Do not use a relative path to the directory.

   Data is imported into the file.

4. Examine the data by selecting the `WorldCupStatistics` table icon and clicking the Data tab.

5. Exit Cloudview.
Virtual Tables, External Data, and Aggregates

NOTE: Cloudscape also provides an export utility. In addition, Cloudview allows you to use alternate file formats for import and export. See the Cloudview on-line help for more information.

External Virtual Tables and External Databases

NOTE: This is an advanced topic.

Cloudscape provides another class, called `COM.cloudscape.vti.ExternalQuery`, that fulfills the VTI requirements, which you learned about in “External Virtual Tables and Bulk Import” on page 8-5. The `ExternalQuery` class uses a JDBC URL and text string representing a query to get data out of any RDBMS for which there is a JDBC driver, including Cloudscape databases. An application that uses this class is responsible for starting the appropriate JDBC driver and for using the correct protocol in the URL to specify the data source.

The `ExternalQuery` class is a good example of how you can write your own class to gain access to external data. You may want to copy or extend this class in your own applications. Cloudscape provides source code for the class in on our Web site at http://www.cloudscape.com/support/Downloads/.

In this section, you will run queries that access data in a Microsoft Access database.

NOTE: You can do this task only if you are running in a Windows environment and only if you have Microsoft Access installed.

Before running the program, you will need to configure the ODBC data source for the database you will be accessing, as described in the following task.
Configure the ODBC Data Source for the Access Database

Accessing ODBC databases requires that you configure a data source for your database using the ODBC Data Source Administrator. You need only configure the data source once in your environment.

<table>
<thead>
<tr>
<th>Quiz: How does this requirement compare to Cloudscape’s configuration requirements?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answer: Cloudscape databases do not require any setup or configuration on a user’s machine; you can deploy them without having to worry about your users’ going through such a step.</td>
</tr>
</tbody>
</table>

1 Choose Settings->Control Panel from the Start menu.
2 Double-click the ODBC icon.
3 Click Add to add a new data source.
4 Select the driver for Microsoft Access and click Finish.
5 For Data Source Name, enter:
   \texttt{CloudscapeTutorial}
6 Under Database, click Select.
7 Navigate to the \texttt{your_tutorial_home} directory (which contains all the files copied from the \texttt{scripts} directory).
8 Select \texttt{Music1.mdb}.
9 Click OK until all the windows are closed.

Load the ODBC-JDBC Bridge Driver and Import Data

1 Start \texttt{ij}, following the instructions in “Start \texttt{ij}” on page 6-6.
2 The Cloudscape JDBC driver should already be running; now you need to start the JDBC driver for ODBC data sources, called a JDBC-ODBC bridge. This driver is a part of the core JDK and is called \texttt{sun.jdbc.odbc.JdbcOdbcDriver}.
3 Use the \texttt{ij} driver command:
Virtual Tables, External Data, and Aggregates

        driver 'sun.jdbc.odbc.JdbcOdbcDriver';

        Quiz: Would this command work in a client/server environment?
        Answer: This command starts the bridge. Cloudscape is the software that needs the bridge. In an embedded environment, this command works, because ij and Cloudscape are sharing the same JVM, so loading the same JDBC driver in the client application loads it in Cloudscape as well. However, ij and Cloudscape would be running in different JVMs. In that situation, you would have to use the following SQL-J statement (which works in embedded mode too):

        VALUES (new sun.jdbc.odbc.JdbcOdbcDriver() IS NOT NULL);

4 Now connect to the HelloWorldDB:

        connect 'HelloWorldDB';

5 Turn off auto-commit:

        autocommit off;

An important use of ij is to run SQL scripts. You can use the Run command to run a script containing a batch of SQL statements.

6 Run a script that creates four tables and imports data from the ODBC data source using ij’s Run command:

        run 'GetODBCData.sql';

ij echoes the scripts as it executes each command, so you will be able to see what is happening. Because we have already loaded the appropriate JDBC driver, the ExternalQuery is able to access data from the Access database by using the URL that you pass as a parameter.

The script created four tables and imported data into those tables from the corresponding tables in the Access database.

7 Test that the data is in the local tables with the following query:

        SELECT RecordingArtist, Title FROM RecordingArtists, Recordings
        WHERE RecordingArtists.artist_id = Recordings.artist_id;

8 Disconnect:

        disconnect;

9 Exit ij:

        exit;

Learning Cloudscape
Aggregate Data

One of the classic reasons for using a DBMS to store and retrieve data is that it makes it very easy to work with aggregates. Aggregates, sometimes also described as *set functions* or *column functions*, provide a means for evaluating an expression over a set of rows. Aggregates can calculate the minimum, maximum, sum, count, and average of an expression over a set of values as well as count rows. Aggregates are what help you analyze and see trends in raw data.

Cloudscape also allows you to create your own aggregates if you want to do something more sophisticated or unusual with a set of data than what the built-in aggregates allow you to do.

The tasks in this section will be more fruitful if you have a bit more data in some of the tables. To this end, run the BuildATour application a few more times now so that you have more tour-related customer data to analyze.

**Run JBM STours.BuildATour a Few More Times**

1. Open a command window and change directories to *your_tutorial_home*.
2. Run your *setclasspath* script (see “Running the Script” on page 2-8).
3. Run *BuildATour*:

   ```
   java -Dcloudscape.system.home=your_tutorial_system
   JBMSTours.BuildATour
   ```
4. Run it three or four times.

**Start ij and Work with Aggregates**

1. Start ij and connect to *toursDB*, following the instructions in “Start ij” on page 6-6 and “Connect to toursDB” on page 6-6.
   The COUNT aggregate function counts the number of rows for which a particular column is not NULL.
2. Count the number of maps in the *Maps* table:

   ```
   SELECT COUNT(map) FROM Maps;
   ```
3. Count the number of cities in Europe:
SELECT COUNT(city) FROM Cities, Countries
WHERE Cities.country_ISO_code = Countries.country_ISO_code
AND Region = 'Europe';

To count the number of rows in a table, use the COUNT(*) expression.

4 Count the number of rows in the HotelAvailability table:

SELECT COUNT(*) FROM HotelAvailability;

5 MAX and MIN return the maximum and minimum value, respectively, of a set of rows. Find the length of the longest flight in the Flights table:

SELECT MAX(miles) FROM Flights;

6 Find the length of the shortest one:

SELECT MIN(miles) FROM Flights;

The aggregate functions work on Java methods, as long as those methods return built-in data types.

7 Find the age of the oldest person in the People table:

SELECT MAX(person.getAge()) FROM People;

8 Find out the largest running total (amount spent so far) in the Groups table:

SELECT MAX(running_total) FROM Groups;

9 Find the average cost of a tour.
AVG calculates the average value of a set of rows. The keyword DISTINCT eliminates duplicate values before calculating the average.

SELECT AVG(customized_tour.getTotalCost()) FROM CustomizedTours;

10 Find the average total mileage of a tour:

SELECT AVG(customized_tour.getTotalMilesTraveled()) FROM CustomizedTours;

11 Find the shortest flight originating from San Francisco:

SELECT MIN(miles) FROM Flights
WHERE orig_airport = 'SFO'
Work with the User-Defined Aggregate MAXBUTONE

toursDB has one user-defined aggregate called MAXBUTONE. It is similar to MAX, except it returns the second highest value in a column instead of the highest value.

User-defined aggregates are backed up by user-supplied Java classes and in that sense are similar to aliases. MAXBUTONE is backed up by the Java class JBMSTours.aggregates.MaxButOneDef. Examine that class to get a glimpse of how this works.

MAXBUTONE works only on columns of type Integer and type Double Precision.

1. Try it on the miles column (type Integer) in the Flights table:

   ```sql
   SELECT MAXBUTONE(distinct miles) FROM Flights;
   ```

   Specifying distinct means that Cloudscape eliminates duplicate values. Otherwise, if there were two values tied for first place, one of those would be returned as MAXBUTONE.

2. Compare that to MAX:

   ```sql
   SELECT MAX(miles) FROM Flights;
   ```

3. Try it on the flying_time column (type double):

   ```sql
   SELECT MAXBUTONE(flying_time) FROM Flights;
   ```

GROUP BY and GenerateReport

SQL allows you to see aggregates for subsets of data. It provides GROUP BY for this. You combine GROUP BY with any aggregate. When combined with the HAVING clause, it allows you to see aggregates for any subsets that meet an aggregate-related search condition.

Use GROUP BY

1. Find the hottest city in each region. You will have to join the Cities and Countries tables.

   ```sql
   SELECT MAX(city.showTemperature()), Region
   FROM Cities JOIN Countries
   ON Countries.country_ISO_code = Cities.country_ISO_code
   GROUP BY Region
   ```
2 Now find the second hottest city in each region. You will have to join the Cities and Countries tables.

```
SELECT MAXBUTONE(city.showTemperature()), Region
FROM Cities JOIN Countries
ON Countries.country_ISO_code = Cities.country_ISO_code
GROUP BY Region
```

**Use HAVING**

1 Find the hottest city in each region, but show only those regions that have an average temperature of at least 70 degrees:

```
SELECT MAX(city.showTemperature()), Region
FROM Cities JOIN Countries
ON Countries.country_ISO_code = Cities.country_ISO_code
GROUP BY Region
HAVING AVG(city.showTemperature()) > 70.0
```

**Quiz:** What’s wrong with the following query?

```
SELECT SUM(rooms_taken) FROM HotelAvailability
GROUP BY booking_date HAVING rooms_taken > 3
```

**Answer:** HAVING should operate on an aggregate-related expression or a grouping column in conjunction with a GROUP BY, not on a column expression. The query can be rewritten to use a WHERE clause instead:

```
SELECT booking_date, SUM(rooms_taken)
FROM HotelAvailability
WHERE rooms_taken > 3
GROUP BY booking_date
```

Here’s a similar query with a correct use of the HAVING clause:

```
SELECT booking_date, SUM(rooms_taken) FROM HotelAvailability
GROUP BY booking_date
HAVING AVG(rooms_taken) > 3
```

2 Exit ij and shut down the system with the exit command:

```
exit;
```
Run JBMSTours.GenerateReport

GenerateReport is an application that uses aggregates like the ones in the previous examples to generate a “report” about the toursDB repository data and about JBMSTours customers.

1. Open a command window and change directories to your_tutorial_home.
2. Run your setclasspath script.
3. Open GenerateReport.java (in /demo/programs/tours/JBMSTours) and look at the queries it runs.
4. Run the program, specifying the system directory with the -D parameter:
   ```
   java -Dcloudscape.system.home=your_tutorial_system
   JBMSTours.GenerateReport
   ```
5. Examine the output.
Virtual Tables, External Data, and Aggregates
NOTE: This chapter covers advanced concepts.

Concepts covered:

- **Transaction management and locking**
  “Transactions” on page 9-1
- Multiple connections to a single database
- Multiple connections to more than one database

Tasks:

- “Start ij and Open a New, Named Connection” on page 9-5
- “Open a Second Named Connection” on page 9-6
- “Begin a Transaction in Each Connection” on page 9-6
- “Commit TransactionA” on page 9-8
- “Repeat the Previous Tasks Using READ_COMMITTED” on page 9-9
- “Get Cloudscape to Automatically Roll Back a Transaction” on page 9-10
- “Run JBMSTours.ArchiveData” on page 9-14

Transactions

Relational DBMSs owe much of their success to their support for transactions. A transaction is a set of one or more SQL statements that make up a logical unit of
work. A transaction ends with either a commit or a rollback (which you saw in “Database-Side JDBC Methods” on page 6-10). A commit makes permanent the changes resulting from the SQL statements in the transaction; rollback undoes them all. Commit and rollback, which are JDBC methods on a Connection object, also mark the beginning of a new transaction.

It is common to talk about the ACID properties of transaction control:

- **Atomic**
  All of the operations in a command are performed, or none of them. The transaction has atomicity: it either commits or aborts.

- **Consistent**
  Transactions allow programmers to declare consistency points that can be validated by the system.

- **Isolated**
  Since more than one transaction can be in process at a time, the system must give each user data that appears unaffected by other transactions until the user’s transaction commits.

- **Durable**
  A transaction’s committed updates must be durable—even in the case of hardware or software failures. Once a transaction commits, it stays committed.

Transactions guarantee *atomicity*, which is useful in the case of system failure. If a system goes down while a transaction is pending, when the system restarts the entire transaction is rolled back. For example, in a banking-related application, a transaction that involves a transfer of money from one account to another might involve two steps:

- withdrawing money from a savings account
- depositing money into a checking account

Without transaction control, if the system crashes between the two steps, the customer has lost money. However, since the steps are part of the transaction, when the system comes up, all transactions that were as yet uncommitted are rolled back. That means that the money is returned to the savings account.

Transactions also guarantee *durability*. This means that transactions, once committed, won’t be affected by system failures. If the above transaction commits before the system goes down, when the system comes up, the database state reflects the activities of the transaction.

Transactions allow a DBMS to maintain *consistency*. 
A DBMS must isolate one transaction from another. In a multi-user system, it is possible for two users to look at the same data at the same time. If one user inserts, updates, or deletes data, it is possible that the other user may see partly old and partly updated data; data will appear to be inconsistent. The degree to which a DBMS isolates one transaction from another is standardized by both SQL-92 and JDBC. Using the JDBC terminology, the isolation levels that Cloudscape currently supports are called:

- **TRANSACTION_SERIALIZABLE**
  When more than one transaction is in process at one time, Cloudscape serializes the transactions. This means that if a user, within a transaction, alters data, all other users are denied access to the data until the transaction has committed. If a user within a transaction views data in a table, all other transactions are prevented from altering the data until the transaction has committed. Even though applications may attempt actions against the database concurrently, Cloudscape processes them as if they happened serially (one after the other). Within the TRANSACTION_SERIALIZABLE level, some actions are permitted to happen concurrently:
  - Two or more transactions may view (SELECT) the same data.
  - If a transaction is the first to view data and other transactions read the data, the first transaction can modify or delete the data; other transactions cannot.

- **READ_COMMITTED** (the default setting)
  When more than one transaction is in process at one time, Cloudscape makes sure that a transaction reads changes made by other transactions only when they have been committed by the transaction. Transactions are permitted to modify data already viewed by other uncommitted transactions, but they are not permitted to modify data that is currently being viewed by uncommitted transactions (the current row). Once a single transaction modifies data, no other transaction can view or modify the data until that transaction commits.

**Transaction Isolation and Locking**

Cloudscape enforces the isolation of transactions with locking. The first transaction to SELECT data gets a shared lock on the data. This lock may be shared with SELECT statements from other transactions.
In order to insert, update, or delete data, a transaction needs an exclusive lock on data. A transaction can get an exclusive lock if it currently has a shared lock on the data, or if no other transaction has any lock on the data.

Once data is locked by an exclusive lock, no other transaction can view (SELECT) or insert, update, or delete the data.

You can configure Cloudscape to use a simple kind of locking called table-level locking. That means that when a transaction gets a lock on data, it locks the entire table, even if only a single row is involved in the transaction, until a commit or rollback releases the lock. Table-level locking uses fewer resources but provides poor performance for multi-user applications (unless they are read-only).

The default setting for Cloudscape is to use a more sophisticated kind of locking called row-level locking. Row-level locking means that a transaction gets a lock only on the rows involved in the transaction, not on the entire table. Such a locking schema allows greater concurrency among transactions but uses more resources than table-level locking. Multiple-user systems typically benefit from row-level locking.

Transactions and Connections

In a Cloudscape system, a transaction is associated with a single connection to Cloudscape. A single connection cannot do the work of more than one transaction; and the work of a single transaction cannot span more than one connection.

Also, a transaction does not endure longer than a single session. A session begins when a connection is first established, and ends when a connection ends.

A single connection can execute multiple transactions only serially—one after the other. However, multiple connections can execute transactions concurrently (at the same time as each other).

In a multi-user system, a connection—and its transactions—are typically associated with a single user. However, because Cloudscape permits an application to have more than one connection to the system, a single application/user may be associated with more than one connection/transaction.

NOTE: The queries in this chapter should return rows if you have run JBMSTours.BuildATour, as described several times in this tutorial. If you skipped those sections, run that application as described in “Run JBMSTours.BuildATour a Few More Times” on page 8-11 before trying out this chapter.
In this section, you will first set the transaction isolation level for each transaction to TRANSACTION.Serializable, the most restrictive. Then you will set the isolation level to READ_COMMITTED and observe the difference in behavior.

**Start ij and Open a New, Named Connection**

ij permits you to name a connection with the `connect` command. In this section, you will open a named connection to `toursDB` and turn off the auto-commit feature.

1. Start ij following the instructions in “Start ij” on page 6-6.
2. Open a connection to `toursDB` called TransactionA. Turn off auto-commit by using the “false” value with the autocommit attribute on the database name.

   ```sql
   connect 'toursDB;autocommit=false' as TransactionA;
   ```
3. Change the isolation level for this transaction to SERIALIZABLE.

   ```sql
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   ```
4. Have ij show all current connections using the `show connections` command.

   ```sql
   show connections;
   ```

   ```sql
   TRANSACTIONA* - jdbc:cloudscape:toursDB;autocommit=false
   * = current connection
   ```

**Set a Database Property**

The default lock wait timeout is 90 seconds. You probably don’t want to wait that long when you do some of the steps in this lesson, so set a property to lower that to 35 seconds. You will do that by setting a database property, which is a property that is stored in the database and valid for the current database only.

To set a database property, you use the void method `setDatabaseProperty` in `COM.cloudscape.databasePropertyInfo` (aliased as `PropertyInfo`). This method takes two arguments:

- the name of the property
- its value

1. Execute the following SQL-J statement:
CALL
PropertyInfo.setDatabaseProperty('cloudscape.locks.waitTimeout', '35');

2  Commit:
    commit;

3  Double-check that you did it right:
    VALUES PropertyInfo.getDatabaseProperty('cloudscape.locks.waitTimeout');

The statement should return 35.

Open a Second Named Connection
1  Open a second connection named TransactionB:
    connect 'toursDB;autocommit=false' as TransactionB;

2  Change the isolation level for this transaction to SERIALIZABLE.
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

3  Have ij show all current connections using ij’s show connections command.
    show connections;

    TransactionA - jdbc:cloudscape:toursDB;autocommit=false
    TransactionB* - jdbc:cloudscape:toursDB;autocommit=false
    * = current connection

Note that the current connection is now TransactionB.

4  Switch back to TransactionA using ij’s set connection command:
    set connection TransactionA;

    An ij command prompt including the name of the current connection in parentheses should appear:
    ij(TransactionA)>

    The connection name appears in the ij prompt only when more than one connection is available.

Begin a Transaction in Each Connection
1  In TransactionA, view all data in the HotelBookings table:
SELECT * FROM HotelBookings;

2 Switch to TransactionB:

set connection TransactionB;

3 View all data in the HotelBookings table:

SELECT * FROM HotelBookings;

Both transactions can view data from this table concurrently. When a transaction selects data from a table, it gets a shared lock on data in the table. This lock can be shared with any other transaction that tries to select data from the table. However, another transaction that tries to modify data will not be able to. In order to INSERT, UPDATE, or DELETE data, a transaction needs an exclusive lock. A transaction cannot get an exclusive lock on a table when another transaction has a shared lock on it.

4 From the current connection (TransactionB), try to insert a row into the table:

INSERT INTO HotelBookings VALUES
(100, 4, DATE'1998-01-01', DATE'1998-01-05', 3, 900.00);

ij will not respond for quite some time (well, for about 35 seconds). Cloudscape is preventing TransactionB from updating data currently being viewed (held in a shared lock) by TransactionA. Since no one is attending to TransactionA to complete that transaction, the system should respond with a lock wait timeout.

ij should eventually respond with an error message:
ERROR 40XL1: A lock could not be obtained within the time requested

TransactionB could not get the exclusive lock. TransactionB’s transaction is rolled back, and a new transaction implicitly begins.

Quiz: I don’t get a deadlock; I can read the data just fine. Why?

Answer: Turn off auto-commit. With auto-commit turned on, a commit is implicitly executed after every statement, freeing up locks.

5 Return to TransactionA:

set Connection TransactionA;

6 Now try to insert the row from within TransactionA:
Working with Connections and Transactions

```
INSERT INTO HotelBookings VALUES
(100, 4, DATE'1998-01-01', DATE'1998-01-05', 3, 900.00);
```

TransactionA is able to promote its shared lock to an exclusive lock and can successfully insert the data.
1 row inserted/updated/deleted

7 From TransactionA, view the current rows in `HotelBookings`:

```
SELECT * FROM HotelBookings;
```

The new row is visible from the current transaction even though it is not yet committed.

8 Switch to TransactionB:

```
set connection TransactionB;
```

9 Try to view data in the table:

```
SELECT * FROM HotelBookings;
```

**Quiz:** Will you be able to view data in the table?

*Answer:* No. TransactionA now has an exclusive, nonshareable lock on data in the table.

**Commit TransactionA**

1 After the transaction times out, return to TransactionA and commit the transaction:

```
set Connection TransactionA;
commit;
```

2 Switch to TransactionB:

```
set connection TransactionB;
```

3 Now try to view data in the table:

```
SELECT * FROM HotelBookings;
```

TransactionB can now view data in `HotelBookings`, and the new (committed) row is visible.

*Learning Cloudscape*
Repeat the Previous Tasks Using READ_COMMITTED

In the previous two tasks, you were working in the TRANSACTION_SERIALIZABLE isolation level, the most restrictive in terms of concurrency. In this task, you will go through some of the same steps in the READ_COMMITTED isolation level to see how the system allows greater concurrency.

1. Commit TransactionB:
   ```
   commit;
   ```

2. Change the isolation level for this transaction to READ_COMMITTED.
   ```
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
   ```

3. Switch to TransactionA, and change its isolation level to READ_COMMITTED also:
   ```
   set connection TransactionA;
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
   ```

4. Select all the data out of the table:
   ```
   SELECT * from HotelBookings;
   ```

5. Switch to TransactionB and try to insert data into the table:
   ```
   set connection TransactionB;
   INSERT INTO HotelBookings VALUES (100, 2, DATE'1998-01-01', DATE'1998-01-05', 3, 900.00);
   ```
   Success! The first time you tried this, Cloudscape did not allow you to insert data into the table. This time, it does. TransactionA does not have a lock on all the data in the table, because it has already finished stepping through the results in the SELECT (in the read committed isolation level).

6. See if TransactionA can see the new row:
   ```
   set connection TransactionA;
   ```
SELECT * FROM HotelBookings;

<table>
<thead>
<tr>
<th>Quiz: Will you be able to view data in the table?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answer: No. TransactionB now has an exclusive, nonshareable lock on data in the table that TransactionA is trying to view.</td>
</tr>
</tbody>
</table>

When TransactionA tries to view all the data, it cannot get a lock on the data it is trying to view.

7 After the transaction times out, try to view only specific rows:

SELECT * FROM HotelBookings WHERE hotel_id > 150;

8 Commit both transactions:

commit;

cset connection TransactionB;

commit;

Get Cloudscape to Automatically Roll Back a Transaction

A transaction that gets a deadlock error receives an SQLException with a transaction severity. Cloudscape automatically rolls back the entire transaction, not just the statement.

1 In TransactionA, insert a row into the Countries table:

    set connection TransactionA;

    INSERT INTO COUNTRIES VALUES ('Bhutan', 'BH', 'Asia');

2 Check to see if the (uncommitted) insert worked:

    SELECT * FROM Countries WHERE Country LIKE 'B%';

    Bhutan should appear. The current transaction is allowed to see its own uncommitted changes.

3 Switch to TransactionB:

    set connection TransactionB;

4 Insert a row in the HotelBookings table:

    INSERT INTO HotelBookings VALUES
        (105, 2, DATE'1998-01-01', DATE'1998-01-05', 3, 900.00);
5 Switch back to TransactionA:

```sql
Set Connection TransactionA;
```

6 Try to view data in the `HotelBookings` table:

```sql
SELECT * FROM HotelBookings;
```

ij will not respond for quite some time (35 seconds). Cloudscape is preventing TransactionA from reading data currently being updated (held in an exclusive lock) by TransactionB. The timeout causes Cloudscape to abort the transaction with a rollback. It therefore implicitly begins a new transaction.

7 Without changing connections, check to see if the first insert (into the `Countries` table) was rolled back:

```sql
SELECT * FROM Countries WHERE Country LIKE 'B%';
```

*Bhutan* should not appear; since that insert statement was part of the transaction that got an error of transaction severity, it was rolled back, along with the rest of the transaction.

## Work With the LockTable VTI

In multi-connection systems, it is sometimes useful to be able to get information about locks. Remember VTIs? (you learned about those in Lesson 8, “Virtual Tables, External Data, and Aggregates”). Cloudscape provides a built-in VTI class, `COM.cloudscape.vti.LockTable` (aliased as `LockTable`) that contains information about locks.

To make things easier, `toursDB` contains a view that selects from this vti. This view is called `Locks`. In this task, you can simply select from the view instead of having to instantiate the VTI.

1 Switch back to transactionB, and commit.

```sql
set connection transactionB;
commit;
```

2 Now switch back to transactionA.

```sql
set connection transactionA;
```

3 Insert a new row into `HotelBookings`:
INSERT INTO HotelBookings VALUES
(102, 2, DATE'1998-01-01', DATE'1998-01-05', 3, 900.00);

4 Now let’s look at the locks on the HotelBookings table. Execute the following SQL-J statement:

SELECT XID, TYPE, MODE, LOCKNAME, STATE, INDEXNAME
FROM LOCKS WHERE TABLENAME = 'HOTELBOOKINGS';

The output should show something like this (the actual transaction XIDs and locknames will be different):

<table>
<thead>
<tr>
<th>XID</th>
<th>Type</th>
<th>Mode</th>
<th>Lockname</th>
<th>State</th>
<th>Indexname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1691</td>
<td>TABLE</td>
<td>IX</td>
<td>Tablelock</td>
<td>GRANT</td>
<td>NULL</td>
</tr>
<tr>
<td>1691</td>
<td>ROW</td>
<td>X</td>
<td>(2,10)</td>
<td>GRANT</td>
<td>NULL</td>
</tr>
</tbody>
</table>

This output shows one lock of type row on the table, held by transaction id 1691 (the current transaction, transactionA). The mode of this lock is X (which stands for exclusive). That makes sense, because the transaction just inserted a single row. (Your output may show two row locks on the table; sometimes inserts lock the previous row as well).

It also shows a lock of type table and mode IX on the table. IX stands for intent exclusive; the transaction does not hold an exclusive lock on the table; this intent lock is there as a placeholder to block any other transactions from getting a table lock on the table.

It also shows an indexname of NULL (meaning no index was used). That makes sense, too, because INSERT statements do not use indexes.

5 Go back to transactionB and update a single row:

`set connection transactionB;`

`UPDATE HOTELBOOKINGS SET DEPARTURE=DATE'1998-01-06' WHERE hotel_ID=105 AND group_id=2 AND ARRIVAL=DATE'1998-01-01';`

6 Now look at all the locks on the HotelBookings table:
SELECT XID, TYPE, MODE, LOCKNAME, STATE, INDEXNAME
FROM LOCKS WHERE TABLENAME = 'HOTELBOOKINGS';

Your output should show something like the following (keeping in mind that it could show one more row lock for the first transaction):

<table>
<thead>
<tr>
<th>XID</th>
<th>Type</th>
<th>Mode</th>
<th>Lockname</th>
<th>State</th>
<th>Indexname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1691</td>
<td>TABLE</td>
<td>IX</td>
<td>Tablelock</td>
<td>GRANT</td>
<td>NULL</td>
</tr>
<tr>
<td>1692</td>
<td>TABLE</td>
<td>IX</td>
<td>Tablelock</td>
<td>GRANT</td>
<td>NULL</td>
</tr>
<tr>
<td>1691</td>
<td>ROW</td>
<td>X</td>
<td>(2,12)</td>
<td>GRANT</td>
<td>NULL</td>
</tr>
<tr>
<td>1691</td>
<td>ROW</td>
<td>X</td>
<td>(2,9)</td>
<td>GRANT</td>
<td>NULL</td>
</tr>
<tr>
<td>1692</td>
<td>ROW</td>
<td>X</td>
<td>(2,19)</td>
<td>GRANT</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Note two new rows for the UPDATE statement: a different transaction id (it's a different transaction); a new IX Tablelock; a new X row lock (on a different row). Your output may or may not show an indexname. The Update statement is eligible to use an index; in our case, the optimizer did not use the index because the table was so small.

7 Rollback both transactions and exit ij.

rollback;
set connection transactionA;
rollback;
exit;

Connecting to Multiple Databases

An application can connect to more than one database in a Cloudscape system. 
*JBMSTours.ArchiveData* is an example of such an application. It executes two methods that select data out of one database, inserts it into another (*History*), then deletes it from the first database (*toursDB*). These methods are the *archiveRecords* and the *archiveRecordsVTI* methods in the *JBMSTours.serializabletypes.HotelStay* class. The methods accomplish the same tasks in different ways.

*ArchiveData* executes the methods as database-side JDBC methods (inside an SQL-J statement), but it could just as easily execute them on the application side. It executes them as database-side methods for illustration purposes only.
Working with Connections and Transactions

The `archiveRecords` method creates two new connections: one to each database. That means that the work accomplished by the method is not encompassed by a single transaction and is not protected by transaction control. It must also handle the results programmatically; it steps through the `ResultSet` from the first database and inserts into the second database. The `archiveRecordsVTI` method, on the other hand, makes use of Cloudscape’s VTI feature (which you learned about in “External Virtual Tables and Bulk Import” on page 8-5) to transfer the data, and so does not need to handle the results programmatically. In addition, for some of the steps, it can re-use the current connection and so can allow some transaction control.

Run JBMSTours.ArchiveData

1. Open a command window and change directories to `your_tutorial_home`.
2. Run your `setclasspath` script.
3. Open and examine `ArchiveData.java` in `/demo/programs/tours/JBMSTours`.
4. Run the program, specifying the system directory with the -D parameter:
   ```
   java -Dcloudscape.system.home=your_tutorial_system
   JBMSTours.ArchiveData
   ```
10 Storing Objects and Classes

Concepts covered:

- **Object storage and serialization**
  “Storing Objects in the Database” on page 10-2
- **Storing subclasses**
  “The People Table and Storing Subclasses” on page 10-7
- **Storing media**
  “Storing Media Objects and Playing Them within Cloudview” on page 10-9
- **Storing classes**
  “Storing Classes in the Database” on page 10-12

Tasks:

- “Create a Class for HelloWorldDB” on page 10-2
- “Try Altering the Class and Storing New Instances” on page 10-5
- “Add the Explicit serialVersionUID to Square” on page 10-6
- “Select Instances of Subclasses from the Person Table” on page 10-8
- “Play the Maps in toursDB” on page 10-10
- “Examine Picture.java to See How the Image Is Stored” on page 10-11
- “Insert Some URLPictures” on page 10-11
- “Create a Jar File for the JBMSTours Package and Subpackages” on page 10-13
- “Add the Jar File to the Database as ToursLogic” on page 10-13
- “Alter the Value of Your Class Path” on page 10-14
Storing Objects and Classes

- “Run JBMSTours.GenerateReport with Class Loading from the Database” on page 10-15

Storing Objects in the Database

Cloudscape allows you to store Java objects in tables. To be storable, an object must be **serializable**. **Serialization** is a Java mechanism for reading and writing objects to a stream; an object is serializable if its class—or any of its superclasses or the interfaces it implements—implements `java.io.Serializable` or `java.io.Externalizable`. When an object is serializable, Java knows how to write the value of each of its fields to an output stream and how to read the value of each of its fields from an input stream.

To define a table to use a Java data type, you use the keyword `SERIALIZE` along with the name of the serializable class or interface to specify the data type:

```plaintext
columnName SERIALIZE( className or aliasName )
```

**NOTE:** Serialization is a relatively slow process. For that reason, it is more efficient to store built-in data types than objects. Where possible, you may want to “translate” an object into built-in types for storage. The `toursDB` database contains many objects to teach you how to work with objects in SQL if you choose to. In reality, the database should probably contain more built-in data types.

Create a Class for HelloWorldDB

In this task, you will create a new class and store instances of it in the `HelloWorldDB` database.

1. Open the file `Square.java`. `Square` is currently an empty class. You will define one field for the class (width), two constructors, and one method (`calculateArea`).

2. Create one field for the width of the square:

   ```java
   int width;
   ```

3. Make two constructors: an empty one that constructs a square of the default size of 10x10, and one that takes one parameter (width).

   ```java
   public Square() {
       width = 10;
   }
   ```
Storing Objects in the Database

4 Give the class a `calculateArea` method (a class without any methods is like a day without sunshine):

```java
public int calculateArea() {
    return width*width;
}
```

The file should look like Figure 10-1.

```java
import java.io.Serializable;
public class Square {
    int width;
    public Square() {
        width = 10;
    }
    public Square(int w) {
        width = w;
    }
    public int calculateArea() {
        return width * width;
    }
}
```

**Figure 10-1**  *Square.java*

5 Save the file and compile the class:

```
javac Square.java
```

6 Start ij following the instructions in “Start ij” on page 6-6.

7 Connect to *HelloWorldDB*:

```
Connect 'HelloWorldDB';
```

8 Try to create a table called *Shapes* with a single column, called *Rectangle*. The data type for the column should be:
SERIALIZE (Square)

So the full table definition should be:

```
CREATE TABLE Shapes (Rectangle SERIALIZE(Square));
```

You should get the following error:

```
ERROR 42X27: The class 'Square' for column 'RECTANGLE' does not implement java.io.Serializable. User-defined types must be serializable.
```

Cloudscape does not let you use the Square data type in the table definition because Square does not implement java.io.Serializable.

9 Exit ij, open Square.java, and edit its definition to implement Serializable:

```
public class Square implements Serializable {

  import java.io.Serializable;

  public class Square implements Serializable {
      int width;
      public Square() {
          width = 10;
      }
      public Square(int w) {
          width = w;
      }
      public int calculateArea() {
          return width * width;
      }
  }
}
```

The file should now look like Figure 10-2.

Figure 10-2  Square.java, stage two

10 Compile the class.

```
javac Square.java
```

11 Start ij and connect to HelloWorldDB.

12 Try again:

```
CREATE TABLE Shapes (Rectangle SERIALIZE(Square));
```

This time, you should see success:

```
0 rows inserted/updated/deleted
```

Learning Cloudscape
13 Insert a few *Squares* just to make sure:

```java
INSERT INTO Shapes VALUES (new Square()),
(new Square(5)), (new Square(10));
```

14 Now try out method invocation:

```java
SELECT MAX(Rectangle.calculateArea()) FROM Shapes;
```

If all goes well, you should see 100 as the result:

```
SQLCol1
---------------
100
```

**Try Altering the Class and Storing New Instances**

1. Exit ij.
2. Open *Square.java* and add a new method:

```java
public int calculatePerimeter() {
    return 4*width;
}
```

The file should now look like Figure 10-3.

![Figure 10-3](Square.java, stage three)

*Cloudscape Version 3.0*
3  Save the file and compile it.
4  Start ij and reconnect to HelloWorldDB.
5  Try selecting from the table:

\[
\text{SELECT * FROM Shapes;}
\]

You should get an error that looks something like the following:
\[
\text{ERROR XSDA8: Exception during restore of a serializable object}
\]
\[
\text{ERROR XJ001: Java exception: 'Square; Local class not compatible: stream classdesc serialVersionUID = -3864197885517086522 local class serialVersionUID = 1230816243124053902: java.io.InvalidClassException'.}
\]
You get this error because Java requires that serialized classes be compatible. Changes in a class automatically violate this compatibility, unless you explicitly tell the JVM that the changes in class are compatible changes. Compatible changes are changes such as new methods. Incompatible changes are changes such as field deletion. Java automatically assumes that any change is an incompatible change unless a class’s serialVersionUID is unchanged. Java includes this automatically generated static final long field in all Serializable classes, regenerating a new value with each permutation of a class. If you explicitly include the value of the field in your class, instead of letting Java do it for you, you have more control over when classes are deemed incompatible, because then you—not Java—decide when the value should change.

In our current situation, the SQLException message has given us a clue: the serialVersionUID of the stored instances of the class is -3864197885517086522. This number may be different in your environment, since it is automatically generated.

**Add the Explicit serialVersionUID to Square**

1  Exit ij.
2  Edit Square.java, adding the following field:

\[
\text{/* paste the actual value from your own error message, and add a capital L at the end of the value to avoid an arithmetic overflow */}
\]
\[
\text{static final long serialVersionUID = -3864197885517086522L;}
\]

The file should now look like Figure 10-4.
The People Table and Storing Subclasses

As you know, toursDB stores several kinds of serializable objects. The People table stores instances of the class JBMSTours.serializabletypes.Person.

It also stores instances of subclasses of the Person data type such as Adult and Child. Cloudscape allows you to store instances of any subclasses of a column’s data type in a column.

The Adult class is identical to the Person class, except that it has an additional method, getPassportNumber (and an additional corresponding private field).
The *Child* class does not have the `getPassportNumber` method, but it has a `getParentId` method (and an additional corresponding private field).

### Select Instances of Subclasses from the Person Table

1. Close the connection to *HelloWorldDB*:
   ```
   disconnect;
   ```
2. Open a connection to *toursDB*:
   ```
   connect 'toursDB';
   ```
3. Turn off auto-commit:
   ```
   autocommit off;
   ```

   You may remember this insert statement from Lesson 6, “Working with SQL-J’s Java Extensions”:
   (Don’t do the insert, just look at it . . .)
   ```
   -- Person is an alias for JBMSTours.serializabletypes.Person.
   INSERT INTO people VALUES (26, new Person('Your', 'Name'));
   ```
4. Try finding out the passport numbers of all the people in the *People* table:
   ```
   SELECT person.getPassportNumber() FROM People;
   ```

   You should get an error message like the following:

   ```
   ERROR 42X50: No method was found with the signature
   JBMSTours.serializabletypes.Person.getPassportNumber().
   It may be that the method exists, but it is not public, or that the parameter types
   are not method invocation convertible.
   ```

   You get the error because the *Person* class (the data type of the column) does not have a method of that signature.

**Quiz:** How do you access that method on the adults?

**Answer:** Cast the data type of the *Person* column to *Adult* (an alias for `JBMSTours.serializabletypes.Adult`).

The CAST operator lets you force one data type to be interpreted as another data type. For Java data types, such casts follow the same rules as explicit casts in Java.
Now try the same query with the CAST operator:

```sql
SELECT (CAST (person as Adult)).getPassportNumber()
FROM People;
```

```
SQLColl
-----------
ABC-DE-FGHX
unknown
ERROR XCL12: An attempt was made to put a data
value of type `JBMSTours.serializatbletypes.Child'
into a data value of type
`JBMSTours.serializatbletypes.Adult'.
```

The table contains instances of the class `Child` as well as the class `Adult`, and a CAST from the `Child` data type to an `Adult` data type is not supported. So to select only the appropriate rows on which to invoke the method, use the INSTANCEOF operator. This operator tests for the data type of an object.

```sql
SELECT (CAST (person AS Adult)).getPassportNumber()
FROM People
WHERE person INSTANCEOF Adult;
```

Now you should be able to get some results.

Exit ij.

Storing Media Objects and Playing Them within Cloudview

Part of the reason for the popularity of the Java programming language is how easy it makes it to integrate media objects like pictures, video, and sound into your applications.

You can use Cloudscape to help organize your application’s media in a number of different ways.

If a class is serializable, you can store it directly in the database. If it is not, you can store the object as a binary (LONG VARBINARY or VARBINARY) data type.

However, you cannot execute any database-side methods on objects stored as non-
Storing Objects and Classes

object binaries. (To execute application-side methods on objects stored in that way, reserialize the object after retrieval.)

If you want to execute database-side methods on objects stored in the database, one solution is to make a serializable thin wrapper class around a byte array field. The examples in this section use that solution.

Media objects such as these aren’t much use unless you have a way to play them. Application developers provide such a way in end-user applications. However, you may want to “play” such objects stored in the database while still in the development phase. If your class has a method that creates a window or dialog box and then displays the object in the window or dialog box, Cloudview lets you “play” these objects.

For example, toursDB has a table called Maps. This table stores objects of type JBMSTours.serializabletypes.Picture. The Picture class has a field called content which stores an image as a byte array. Its display() method creates a new Image from that byte array and displays it in a window, so it can be displayed by Cloudview.

Play the Maps in toursDB

1. Start Cloudview as described in “Start Cloudview in the Home Directory” on page 3-3. Then open a connection to toursDB as described in “Open a Connection to toursDB” on page 3-9. Select the toursDB icon in the left-hand hierarchy window. Click the “+” to the left of the icon to display the Table icon.
2. Click the plus sign to the left of the Table icon to display the tables in toursDB.
3. Select the Maps table in the left-hand hierarchy window.
4. Select the Data tab.
5. The third column in the table, map, is the one that stores the picture. Select the map column in the last row. Click the magnifying class icon. The Object Inspector window appears.
7. Select display and click Execute. The map appears in a new frame.
8. Select any other picture to display it.
Examine Picture.java to See How the Image Is Stored

1. Open the source file for Picture, Picture.java in /demo/programs/tours/JBMSTours/serializabletypes.

2. Examine the class definition.
   One of the fields in the class is an array of bytes. When displaying the image, the class creates a new instance of a java.awt.Image, reading in the data from the byte array field.

**Quiz:** Why can’t you just store a java.awt.Image?

**Answer:** Because java.awt.Image is not serializable. All Java data types stored in a Cloudscape database must be serializable.

Insert Some URLPictures

JBMSTours.serializabletypes.Picture loads a picture available to the JVM on the class path. Sometimes you want to work with media in various places, such as on the Internet. You may want to store the actual media, or you may want to store only references to the media.

JBMSTours.serializabletypes.URLPicture is a subclass of Picture that gets its content from a URL. It contains a static field called MAXSIZE. If the content is larger than that field, the class stores only the URL as a reference. If the content is smaller than that field, the class gets the entire content from the URL and stores that. The display() method is implemented to allow Cloudview to display the picture in either case.

1. Go to the SQL window.

2. Insert a “small” URLPicture (in which the content is actually stored) from the Internet:
   ```sql
   INSERT INTO MAPS VALUES ('AU', false, new JBMSTours.serializabletypes.URLPicture('http://www.lib.utexas.edu/Libs/PCL/Map_collection/australia/Australia_sm97.gif',
   'Small map of Australia'));
   ```
   You can store the URLPicture in the Maps table because URLPicture is a subclass of Picture.

3. Insert a “large” image (in which only a reference is stored) from the Internet:
Storing Objects and Classes

```
INSERT INTO MAPS VALUES ('RU', true, new
JBMSTours.serializabletypes.URLPicture('http://
www.lib.utexas.edu/Libs/PCL/Map_collection/commonwealth/
Russia.94.jpg', 'Big map of Russia'));
```

4 Play these new images. Select the Maps table in the left-hand hierarchy window.

5 Select the Data tab.

6 The third column in the table, map, is the one that stores the picture. Select the map column in the row that contains one of the new pictures. Click the Magnifying Glass icon. The Object Inspector window appears.

7 Select MAP=JBMSTours.serializabletypes.URLPicture. A list of available methods appears in the right-hand window.

8 Select display().

9 Click Execute. The map appears in a new frame.

10 Exit Cloudview.

Storing Classes in the Database

**NOTE:** This is an advanced topic.

You can store application logic in a database and then have Cloudscape and your application load classes from the database instead of from the user’s class path. Application logic, which can be used by both Cloudscape and your application, includes Java class files and other resources. In the JBMSTours application, the City class is an example of application-side logic that is used by both Cloudscape and the application.

Storing application code simplifies application deployment, because it reduces the potential for problems with a user’s class path.

To store classes in a database, you first must package up the class files (and any resources used by the application) into a jar file. Then you load the jar file into the database with a special utility and enable Cloudscape to load classes from a database. Finally, you use a special bootstrap mechanism for executing the application so that it uses the Cloudscape class loader.
In this section, you will package up the class files in the \textit{JBMSTours} package into a jar file and try to run the application using the Cloudscape class loader. You will no longer need \texttt{%CLOUDSCAPE\_INSTALL%/demo/programs/tours} in your class path to run the \textit{JBMSTours} application, because all application logic will now be stored in the database.

**Create a Jar File for the JBMSTours Package and Subpackages**

You can use the standard jar utility to create a jar file for the \textit{JBMSTours} package.

1. Open a command window.
2. Change directories to %CLOUDSCAPE\_INSTALL%/demo/programs/tours.
   
   You must be in the directory that contains the package in order to preserve the package names within the jar file.
3. Issue the jar command to package up all the class files in \textit{JBMSTours} into a jar file called \texttt{tours.jar}. \texttt{tours.jar} will be the jar file’s physical name (it will have another name within Cloudscape).
   
   This command jars up all .class files in the JBMSTours package directory and all subpackages.

   \begin{verbatim}
   jar cf tours.jar JBMSTours/*.class JBMSTours/serializabletypes/*.class JBMSTours/inserters/*.class JBMSTours/vti/jdbc1_2/*.class JBMSTours/triggers/*.class JBMSTours/aggregates/*.class
   \end{verbatim}

   **NOTE:** The jar utility recognizes forward slashes on all platforms.
4. Move this jar file to the \textit{your\_tutorial\_home} directory.
5. Change directories to the \textit{your\_tutorial\_home} directory.

**Add the Jar File to the Database as ToursLogic**

You will add the jar file to the \textit{toursDB} database, giving it a Cloudscape name when you do so. This name will be \textit{ToursLogic}. You will modify it with \textit{APP}, the schema name.

Cloudview makes it easy to work with jar files.

1. Start Cloudview as described in “Start Cloudview in the Home Directory” on page 3-3. Then open a connection to \textit{toursDB} as described in “Open a Connection to toursDB” on page 3-9.
2. Select the Jar Files icon.
3 Choose New->Jar file from the Edit menu.
4 Type ToursLogic in the Name box.
5 APP is the correct schema.
6 Click on the navigate button to find tours.jar, then select it and click Open.
7 Make sure the add to classpath button is selected.
   This enables database class loading for the jar file by setting a database property.
8 Click OK.
   A copy of the jar file is now stored in the database under name ToursLogic.
9 Exit Cloudview.

Alter the Value of Your Class Path

When loading classes, Cloudscape always looks in the class path first, then in the jar files stored in the database. To make sure Cloudscape loads files from the database, remove the %CLOUDSCAPE_INSTALL%/demo/programs/tours directory from your class path.

1 Open the setclasspath script that you learned about in “Customize Your setclasspath Script” on page 2-8 and that you have been running periodically throughout this tutorial.
   - Windows users should open setclasspath.bat.
   - UNIX users should open setclasspath.sh.
   - UNIX users who are working in cshell should open setclasspath.csh.
2 Alter the script by removing %CLOUDSCAPE_INSTALL%/demo/programs/tours from the command that sets your class path.
3 Save the script.
4 Run the script as instructed in “Running the Script” on page 2-8.
5 Verify that you actually removed the appropriate package by trying to run one of the applications:

   java JBMSTours.GenerateReport

   You should get the dreaded “can’t find class” message, but this time, that’s what you want!

   Can’t find class JBMSTours/GenerateReport
Run JBMSTours.GenerateReport with Class Loading from the Database

Once you enable database class loading (which you did in “Add the Jar File to the Database as ToursLogic” on page 10-13) and remove loadable classes from the class path, Cloudscape will load the classes from the database. Cloudscape provides an application bootstrap program that forces Cloudscape class loading for your application. The program is called COM.cloudscape.util.DBClassLoad.

1. Start JBMSTours.GenerateReport with the bootstrap program:

```
java -Dcloudscape.system.home=your_tutorial_system
   COM.cloudscape.util.DBClassLoad jdbc:cloudscape:toursDB
   JBMSTours.GenerateReport
```
11 Servers, Servlets, and Applets

Concepts covered:
- Servers
- Servlets
- Applets

Tasks:
- “Start RmiJdbc Server” on page 11-2
- “Run ij as a Client Application” on page 11-3
- “Open a Client Window, and Run an Application” on page 11-4
- “Configure Cloudconnector for the Servlets and Start Cloudconnector” on page 11-6
- “Configure Java Web Server 1.1 for the Servlet and Start Java Web Server” on page 11-8
- “Use the Servlets” on page 11-11
- “Configure and Start Cloudconnector” on page 11-13
- “Access the Applet from a Browser” on page 11-15

NOTE: You can do this lesson as a stand-alone demo. To do that, first do Lesson 2, then Lesson 5 before trying this one.
Servers, Servlets, and Applets

JDBC Servers

As you learned in “Cloudscape’s Server Frameworks” on page 5-3, only one application can access Cloudscape at a time if Cloudscape is embedded in an application. If you embed Cloudscape in special server framework software, multiple applications can connect to the same database at the same time. These applications connect over the network.

A JDBC server allows applications to execute JDBC calls remotely against a remote JDBC driver.

In this section, you start up RmiJdbc, a lightweight server framework included with the product, then run two client applications.

Start RmiJdbc Server

1. Open a command window, and change directories to your_tutorial_home directory.
2. Run the classPathScript.
3. Add one more library to class path, %CLOUDSCAPE_INSTALL%/frameworks/RmiJdbc/classes/RmiJdbc.jar
   On a Windows platform, you would do that like this:
   ```
   SET CLASSPATH=%CLOUDSCAPE_INSTALL%/frameworks/RmiJdbc/classes/RmiJdbc.jar;%CLASSPATH%
   ``
4. Use a Cloudscape utility that tests your class path for a particular environment. You will use it with the arguments RmiServer, to test for an RmiJdbc server environment, and sampleApp, to test for the sample application libraries.
   ```
   java COM.cloudscape.tools.sysinfo -cp RmiServer sampleApp
   ```
   The utility displays a message indicating success or failure. If you got a failure message, check your class path carefully.
5. Start the server with the following command, being sure to set the value of `cloudscape.system.home` to the location of the directory that contains `toursDB`:

Learning Cloudscape
java -ms32M -mx32M -Dcloudscape.system.home=your_tutorial_system RmiJdbc.RJdjdbcServer COM.cloudscape.core.JDBCDriver

You should get a startup message that looks something like this:

Thu Sep 23 13:20:34 PDT 1999: [RmiJdbc] COM.cloudscape.core.JDBCDriver registered in DriverManager
Thu Sep 23 13:20:34 PDT 1999: [RmiJdbc] Binding RmiJdbcServer...
Thu Sep 23 13:20:34 PDT 1999: [RmiJdbc] No installation of RMI Security Manager...
Thu Sep 23 13:20:35 PDT 1999: [RmiJdbc] RmiJdbcServer bound in rmi registry

The RmiJdbc server is now started.

**Run ij as a Client Application**

1. Open a command window, and change directories to your_tutorial_home directory.
2. Run the *classPathScript*.
3. Add two more libraries to class path, client.jar and %CLOUDSCAPE_INSTALL%/frameworks/RmiJdbc/classes/RmiJdbc.jar
   On a Windows platform, you would do that like this:
   
   ```
   SET CLASSPATH=%CLOUDSCAPE_INSTALL%\lib\client.jar;%CLOUDSCAPE_INSTALL%\frameworks\RmiJdbc\classes\RmiJdbc.jar;%CLASSPATH%
   ```
4. Use a Cloudscape utility that tests your class path for a particular environment. You will use it with the arguments *RmiClient*, to test for an RmiJdbc server environment, and *sampleApp*, to test for the sample application libraries.
   
   ```
   java COM.cloudscape.tools.sysinfo -cp RmiClient sampleApp
   ```
   The utility displays a message indicating success or failure. If you got a failure message, check your class path carefully.
5. Start ij:
   
   ```
   java COM.cloudscape.tools.ij
   ```
Connect to `toursDB` using the correct client database connection URL. To do that, use the protocol `jdbc:cloudscape:rmi:` followed by the host name (`localhost`) and port number (1099 is the default).

```
Connect 'jdbc:cloudscape:rmi://localhost:1099/toursDB';
```

Keep the connection open for now.

### Open a Client Window, and Run an Application

The JBMSTours applications also allow you to run them as a client to RmiJdbc.

1. Open another command window, and change directories to `your_tutorial_home` directory.
2. Set the class path for an RmiJdbc client environment as described in “Run ij as a Client Application” on page 11-3.
3. Run the `classpathScript`.
4. Try running `JBMSTours.Generate Report` as a client application. Do do that, you need to add three arguments:
   - the host name
   - the port number
   - the letter “r” if the server framework is RmiJdbc.
   Since you are running the server framework locally and are using the default port numbers, the host name is `localhost` and the port number is 1099.
   That means you can run the application like this:
   ```
   java JBMSTours.GenerateReport localhost 1099 r
   ```
   Note that you do not need to set the location of `cloudscape.system.home`. You do that on the server only.
5. While the application is running, go back to the first client window.
6. Execute the following SQL-J statement:
   ```
   SELECT * FROM Locks WHERE TableName NOT LIKE 'SYS%';
   ```
7. If you have time, execute it again. Notice that you can see the locks held by the other application.
8. Exit ij:
   ```
   exit;
   ```
From the same client window, stop the server like this:

```
java RJPing jdbc:rmi:jdbc:cloudscape: shutdown
```

Close all windows.

Database Servlets

The JBMSTours application includes two (related) servlets. A servlet is a “server-side” Java application that runs embedded in a Web server. Servlets can start up and access Cloudscape as an embedded Java application.

If you have one of the following Web servers, you can run the servlets:

- Cloudconnector
- Java Web Server 1.1

This lesson is a quick-start on running the example servlets.

Refer to your Web server or application server documentation for detailed information on how to configure and enable servlets. The basic idea is that once you register the servlets correctly, you should be able to access them and see what they do.

There are also several books available on how to create servlets.

About the Servlets

The JBMSTours application includes the following classes:

- `JBMSTours.servlets.OpeningServlet`
  A servlet that generates an HTML page with a form that allows users to choose a city and a range of dates. When users click submit, it submits the information to `GetHotelsForCity`.

- `JBMSTours.servlets.GetHotelsForCity`
  A servlet that receives input parameters from `OpeningServlet` and that displays hotels and availability information for a given city and range of dates.

- `JBMSTours.servlets.HTMLUtil`
  A utility class that supports the HTML generation of the servlets.
Appropriate Web Servers and Web Browsers

If you have purchased and installed Cloudconnector, use that as your Web server. Otherwise, you can use Java Web Server (version 1.1).

You can use any Web browser that supports JavaScript.

Running the Servlets

- Configure Cloudconnector for the Servlets and Start Cloudconnector
- Configure Java Web Server 1.1 for the Servlet and Start Java Web Server

Configure Cloudconnector for the Servlets and Start Cloudconnector

NOTE: These instructions assume that you are familiar with how to start Cloudconnector from the command line. For a good introduction to how to start Cloudconnector, see the instructions for running Cloudscape’s simple demo. You will need to restart Cloudconnector in this example.

1. Open a command window.
2. Change directories to your weblogic.system.home directory, the directory in which you keep the CloudscapeServer folder and the weblogic.properties file. The default location for these is %CLOUDSCAPE_INSTALL%/frameworks/cloudconnect.
3. In this window, set the class path to include:
   - the Cloudscape main library
   - the Cloudconnector libraries (those libraries are described in Getting Started with Cloudscape and in the instructions for the simple demo).
   - the %CLOUDSCAPE_INSTALL%/demo/programs/tours directory

   For example, on a Windows platform, you would do that like this:

```
SET CLASSPATH=%CLOUDSCAPE_INSTALL%/lib/cloudscape.jar;%CLOUDSCAPE_INSTALL%/frameworks/cloudconnect/classes;%CLOUDSCAPE_INSTALL%/frameworks/cloudconnect/lib/weblogicaux.jar;%CLOUDSCAPE_INSTALL%/frameworks/cloudconnect/license;%CLOUDSCAPE_INSTALL%/demo/programs/tours
```
4 Use a Cloudscape utility that tests your class path for a particular environment. You will use it with the arguments CloudconnectorServer, to test for a Cloudconnector server environment, and sampleApp, to test for the sample application libraries.

```java
java COM.cloudscape.tools.sysinfo -cp CloudconnectorServer sampleApp
```

The utility displays a message indicating success or failure. If you got a failure message, check your class path carefully.

5 Edit the weblogic.properties file that resides the current directory. Make the changes described in the following step.

6 Register the servlets provided with this example by adding the following lines to the file at the end of the section called “USER-WRITTEN AND DEMO SERVLET REGISTRATIONS.”

```properties
weblogic.httpd.register.OpeningServlet=
  JBMSTours.servlets.OpeningServlet
weblogic.httpd.register.GetHotelsForCity=
  JBMSTours.servlets.GetHotelsForCity
```

7 Save the file.

8 Start Cloudconnector from the directory that contains the weblogic.properties file that you just edited. When starting Cloudconnector, being sure to set the value of cloudscape.system.home to the location of the directory that contains toursDB: If you’re doing just this lesson, use the pre-built database; simply set the value of cloudscape.system.home to %cloudscape_install%/demo/databases. Start Cloudconnector like this:

```java
java -ms16m -mx32m -Dcloudscape.system.home=your_tutorial_system COM.cloudscape.core.CloudscapeServer
```

9 Start a browser from the same machine or any machine on the network.

10 Open the following location:

```
http://<servername>:<portnumber>/OpeningServlet
```

replacing <servername> and <portnumber> with the correct server name and port number. Most likely, these will be localhost and 7001.

Skip ahead to “Use the Servlets” on page 11-11.
Configure Java Web Server 1.1 for the Servlet and Start Java Web Server

NOTE: These instructions assume that you are familiar with how to start Java Web Server and how to include libraries in its class path. You will need to restart Java Web Server in this example.

1. Before starting the Java Web Server, add the main Cloudscape library to its class path (cloudscape.jar or cloudsync.jar). You can do that by setting the class path in the command window from which you start the server.

2. Copy the entire toursDB directory into the outermost directory for the Java Web Server. (You have to copy the database into the Web server’s working directory because there is not a reliable way of setting a system property for the servlet, at least as of Java Web Server 1.1. For that reason, the servlet always thinks of its working directory as the cloudscape.system.home directory.)

3. Start the Web server.
   c:\JavaWebServer1.1\bin\httpd

4. Register the servlets like this:
   - Start the Administration Tool by clicking on the appropriate link from the main page for the Web server (http://localhost:8080).
   - Select Web server, and click Manage.
   - Click the Servlets icon.
   - Click Add under the servlets hierarchy.
   - Register OpeningServlet. Specify OpeningServlet as the servlet name, and JBMSTours.servlets.OpeningServlet as the servlet class as shown below.
- Click Add.
- Provide a description.
- Click the “no” button next to Load at Startup.
- Click Save.
- Click Add under the servlets hierarchy.
- Now configure GetHotelsForCity. Specify GetHotelsForCity as the servlet name, and JBMSTours.servlets.GetHotelsForCity as the servlet class.
- Click Add.
- Provide a description.
- Click the “no” button next to Load at Startup.
- Click Save.

5 From a browser (same machine or another one on the network), open the following document:

http://<serverName>:<portNumber>/servlet/OpeningServlet

replacing <serverName> and <portNumber> with the correct server name and port number. Most likely, these will be localhost and 8080.

6 Do all the steps outlined in “Use the Servlets” on page 11-11.
Use the Servlets

Once the HTML page is loaded, it provides a window that allows you select a city and a range of dates.

1. If you wish, examine the source code for OpeningServlet in /demo/programs/tours/JBMSTours/servlets/OpeningServlet.java. The servlet queries the database and dynamically generates the HTML.

2. Choose a city, and select a range of dates, then click Submit Query. OpeningServlet sends the parameters you have selected to GetHotelsForCity.

3. If you wish, examine the source code for GetHotelsForCity in /demo/programs/tours/JBMSTours/servlets/GetHotelsForCity.java. The servlet queries the database and dynamically generates the HTML.
The first time you access the servlet, the servlet attempts to boot Cloudscape if it is not already booted. It then attempts to connect to toursDB. (toursDB must be in the cloudscape.system.home directory).

4 Close the Web browser.

5 Stop the server if appropriate.

Working with a Database Applet

NOTE: You can do this example if you purchased and installed Cloudconnector.

Overview

The JBMSTours application includes an applet. An applet is a “client-side” Java application that runs embedded in a Web browser. The applet uses the Cloudconnector client driver to connect to Cloudscape running in the
Cloudconnector framework. Cloudconnector provides both the HTTP and database services; it is both a Web server and a database server.

**NOTE:** The applet runs only in Web browsers supporting at least JDK1.1.4.

No instructions are provided for using appletviewer.

### About the Applet

The JBMSTours application includes the class `JBMSTours.applet.ClientApplet`. This applet starts the Cloudconnector client driver, opens a connections to the `toursDB` database, and provides a SQL query window for the connection with the first suggested query already filled in.

If you wish, examine the source code for the applet in `/demo/programs/tours/JBMSTours/applets/ClientApplet.java`.

The directory `/demo/programs/tours/JBMSTours` also contains a supporting HTML page, `ClientApplet.html`.

### Configure and Start Cloudconnector

**NOTE:** These instructions assume that you are familiar with how to start Cloudconnector on the command line. You will need to restart Cloudconnector in this example. For a good introduction to how to start Cloudconnector, see the instructions for running Cloudscape’s simple demo. You will need to restart Cloudconnector in this example.

1. Stop Cloudconnector if it is already running.
2. Open a command window.
3. Change directories to your `weblogic.system.home` directory, the directory in which you keep the `CloudscapeServer` folder and the `weblogic.properties` file. The default location for these is `%CLOUDSCAPE_INSTALL%/frameworks/cloudconnect`.
4. In this window, set the class path to include:
   - the Cloudscape main library
   - the Cloudconnector libraries (those libraries are described in *Getting Started with Cloudscape* and in the instructions for the simple demo).
   - `client.jar`
   - the `%/CLOUDSCAPE_INSTALL%/demo/programs/tours` directory.
5 Use a Cloudscape utility that tests your class path for a particular environment. You will run it once with the arguments CloudconnectorServer, to test for a Cloudconnector server environment, and sampleApp, to test for the sample application libraries. You will run it again with the argument CloudconnectorClient, to test for a Cloudconnector client environment (the server serves up client classes to the applet remotely).

```java
java COM.cloudscape.tools.sysinfo -cp CloudconnectorServer sampleApp
java COM.cloudscape.tools.sysinfo -cp CloudconnectorClient
```

The utility displays a message indicating success or failure. If you got a failure message, check your class path carefully.

6 Edit the `weblogic.properties` file as follows:

Register WebLogic’s ClasspathServlet so that you can load HTML pages and other resources from the class path.

Give it the virtual name CloudscapeExamples. To do that, search in the file for `weblogic.servlet.ClasspathServlet`. Copy the line and paste it below the current line. In the new text change classes to read CloudscapeExamples. It should read:

```properties
weblogic.httpd.register.CloudscapeExamples=\nweblogic.servlet.ClasspathServlet
```

7 If you already did the servlets lesson, comment out the lines that registered the two servlets, OpeningServlet and GetHotelsForCity.

8 Save the file.

For this example, do not run Cloudconnector in JDK1.2. Use any 1.1 JDK except JDK1.1.7. The applet does not work if Cloudconnector is running in JDK 1.2.

9 Start Cloudconnector on the command line from this directory, like so:

```java
java -ms16m -mx32m -Dcloudscape.system.home=your_tutorial_system COM.cloudscape.core.CloudscapeServer
```
Access the Applet from a Browser

The applet used in this demo is a JDK1.1 applet. It requires support and fixes made in JDK1.1.4 to operate in a Web browser. You can use either of the following browsers:

- Microsoft Internet Explorer 4.0.1. or later
- Netscape Navigator 4.0.4 with the JDK1.1.4 patch applied ONLY. Later versions of Netscape MAY NOT WORK.

The browsers do not need any classes in their class path for this demo. The applet will download all needed classes from the Web server.

NOTE: If you are using Netscape Navigator, your system CLASSPATH must not contain any of the classes the applet will download from the server. It is safest to clean out the system class path before starting the browser.

1. Start a browser on the same machine as the server, or any machine on the network.
2. Open the following location:

   http://<serverName>:<portNumber>/CloudscapeExamples/JBMSTours/applets/ClientApplet.html

   replacing <serverName> and <portNumber> with the correct server name and port number. Most likely, these will be localhost and 7001.

Once the HTML page is downloaded, it loads the applet that provides a window that allows you to send an SQL-J statement to Cloudscape. The results will appear in the new window. The page comes preloaded with a query of the Cities table.
3 Click the Execute Statement button to execute the statement.

4 You can execute any of the queries you’ve learned throughout this tutorial.

**NOTE:** The applet only supports one statement per execution; do not send batches. Security restrictions require that the applet access a database server on the same machine as the Web server that served it up. In this example, Cloudconnector is both the Web server and the database server, so access is not a problem. If you want to test the applet from behind a firewall, you will need to set up Cloudconnector for HTTP tunneling.
Appendix A

Troubleshooting

Common Problems

This appendix helps you get past typical roadblocks.

- “Environment Space Errors in Windows 95” on page A-2
- “I/O Errors on Solaris” on page A-3
- “ClassNotFound When Accessing Demo Database” on page A-4
Environment Space Errors in Windows 95

When setting environment variables such as CLASSPATH in Windows 95 (as in CLASSPATH), users sometimes get a message such as “Not enough environment space.”

To fix this, enlarge the environment space. To enlarge the environment space, set the following command in the CONFIG.SYS file:

```
SHELL=C:\Windows\COMMAND.COM /P /E:4096
```

COMMAND.COM is found in the Windows installation directory (usually C:\windows).

- /P means to leave COMMAND.COM running between commands (i.e, don’t exit when the user issues the dir command).
- /E: signifies how much environment space to reserve for each process. 4K is usually enough.
I/O Errors on Solaris

Customers running Cloudscape on Sun Solaris sometimes experience I/O errors from Cloudscape on Solaris. The cause seems to be an inadequate allocation of file descriptors. Apparently the default is 64 file descriptors per process, and this is inadequate for any software, such as Cloudscape, that needs to open a large number of files.

The definitive information on changing this resource is in the *Sun System Administration Guide for Solaris*, but there are some things any user can do to adjust this limit temporarily from the command line without having root privileges. These measures involve the *limit* command, which differs slightly from shell to shell. Please see the “*limit*” man page (in the Solaris commands reference manual) for more information.

The following commands can be used from the C shell (csh) to change the number of file descriptors allowed in the current shell. (In each example, the “%” is the C shell’s command prompt.)

1. Use the *limit* command to find the current value:

   ```
   % limit descriptors
   descriptors      64
   (64 file descriptors per process is almost certainly not enough for any serious Java work.)
   ```

2. Use the *unlimit* command (csh) to remove the descriptor limit for the current process, then repeat the *limit* command to see that the number of descriptors has increased.

   ```
   % unlimit descriptors
   % limit descriptors
   descriptors      1024
   ```
Troubleshooting Common Problems

ClassNotFound When Accessing Demo Database

Users sometimes get `ClassNotFoundException` errors when trying to access some of the data in the demo database, `toursDB`. The errors look something like this:

```
Database access resulted in SQL exception:
SQLState XSDA9, Error code 20000, Message:
Class not found during restore of a serializable object
SQLState XJ001, Error code 0, Message:
Java exception: ‘JBMSTours.City:
java.lang.ClassNotFoundException’.
```

This is a classic class path error. You can’t access `City` objects because the class definition for `JBMSTours.City` is not in your class path.

Be sure to add the `/demo/programs/tours` subdirectory in the cloudscape base directory to your class path. See “Getting Acquainted with Class Path” on page 2-2.
### Cloudscape Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACID</strong></td>
<td>The four properties of transaction control: Atomicity, Consistency, Isolation, and Durability.</td>
</tr>
<tr>
<td><strong>Agreement</strong></td>
<td>A situation in which the only difference between the corresponding data in two databases is latency. See Latency.</td>
</tr>
<tr>
<td><strong>Applet</strong></td>
<td>A Java program that is executed by a browser, as opposed to a stand-alone Java application.</td>
</tr>
<tr>
<td><strong>Atomicity</strong></td>
<td>The A in ACID. The property of transaction control that means that either all or none of the statements in a transaction are executed.</td>
</tr>
<tr>
<td><strong>Auto-commit mode</strong></td>
<td>A state within a JDBC application that means each individual SQL statement is treated as a transaction and is automatically committed when executed. To group two or more statements into a transaction, you must turn off auto-commit mode in the application, then explicitly commit a transaction. In a Cloudscape application, auto-commit causes cursors to close.</td>
</tr>
<tr>
<td><strong>Backing index</strong></td>
<td>An index that Cloudscape creates to enforce a primary key, unique, or foreign key constraint. So called because the index helps “back up” the constraint. Indexes created in this way are no different from indexes that you create with explicit CREATE INDEX commands, except that they have system-generated names.</td>
</tr>
<tr>
<td><strong>Bulk import</strong></td>
<td>Populating a table by importing data from a flat file instead of through insert statements.</td>
</tr>
<tr>
<td><strong>Bulk export</strong></td>
<td>Exporting the contents of an entire table or any subset of data from a database to a flat file using a Cloudscape-provided utility.</td>
</tr>
<tr>
<td><strong>Check constraint</strong></td>
<td>A boolean test applied to each row of a table on an insert or update to validate it.</td>
</tr>
<tr>
<td><strong>Client/server environment</strong></td>
<td>When multiple applications connect to Cloudscape over the network, they are said to run in a client/server environment. Cloudscape runs embedded in a server framework that allows multiple network connections. (The server framework itself starts up an instance of Cloudscape and, strictly speaking, it is running in an embedded environment; the client applications, however, are not.)</td>
</tr>
<tr>
<td><strong>Class path</strong></td>
<td>A list of directories or zip files to search for class files.</td>
</tr>
<tr>
<td><strong>Commit</strong></td>
<td>An operation that completes a transaction and causes all updates made during the transaction to be permanently written into a database.</td>
</tr>
</tbody>
</table>
### Cloudscape Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Concurrency</strong></td>
<td>Multiple users or programs simultaneously sharing the same database. Transactions and data locks are used to give each of these a consistent view of the database.</td>
</tr>
<tr>
<td><strong>Conflict</strong></td>
<td>In Cloudscape synchronization, a change submitted by a target database that, if allowed to apply at the source, would corrupt the data.</td>
</tr>
<tr>
<td><strong>Conglomerate</strong></td>
<td>A unit of storage within a Cloudscape database, either a table or an index.</td>
</tr>
<tr>
<td><strong>Connection</strong></td>
<td>A session with a database opened by a JDBC application program. Only one transaction at a time can be associated with a connection.</td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
<td>The C in ACID. Consistency means that each transaction starts in a consistent state and ends in a consistent state. One transaction cannot “see” the changes caused by another transaction until that other transaction completes.</td>
</tr>
<tr>
<td><strong>Constraint</strong></td>
<td>In a database, a rule that ensures the integrity of data. Constraints can apply to columns or tables. An example of a column-level constraint is that the values it contains must be unique.</td>
</tr>
<tr>
<td><strong>Cursor</strong></td>
<td>A construct used to reference the current position in a result. SQL defines a cursor mechanism that can be used through JDBC to do positioned updates and deletes.</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>A Cloudscape database stores data as SQL-92 data types and Java objects. A Cloudscape database contains dictionary objects that organize the data. Cloudscape databases are stored in files in folders named after the database.</td>
</tr>
<tr>
<td><strong>Database-side method</strong></td>
<td>A Java method invoked in SQL-J and executed by Cloudscape.</td>
</tr>
<tr>
<td><strong>Database-side JDBC method</strong></td>
<td>A Database-side method that executes SQL-J statements, usually using a nested connection.</td>
</tr>
<tr>
<td><strong>Database class loading</strong></td>
<td>When you store application logic (in the form of jar files) in a database, you can configure Cloudscape (and your application) to load classes from the database instead of from the user’s class path.</td>
</tr>
<tr>
<td><strong>Database connection URL</strong></td>
<td>In JDBC, a URL that you specify as an argument to the DriverManager.getConnection method call and that returns a connection to a database. Cloudscape’s database connection URL allows you to specify attributes to the connection.</td>
</tr>
<tr>
<td><strong>DBMS</strong></td>
<td>Database management system. Software that manages databases. Cloudscape is a DBMS, and it can run as an embedded database system under the same JVM as the database application program, or it can run as a database server, under a different JVM, or even on a different machine from that of the application.</td>
</tr>
</tbody>
</table>
**DDL**

Data Definition Language. A subset of SQL commands that create dictionary objects.

**Dependency**

The relationships among prepared statements and dictionary objects in a database. Cloudscape tracks the dependency that publications, stored prepared statements, and prepared statements have on the dictionary objects they access. If dropping or modifying a dictionary object would cause an open prepared statement to be invalidated, the change is not permitted.

**Dictionary object**

A user-defined entity contained in a Cloudscape database. When creating a database, users create and name dictionary objects such as tables, indexes, constraints, views, and stored prepared statements. Information about these dictionary objects is stored in the Cloudscape system tables, sometimes called the data dictionary.

**Disk encryption**

Encrypting a Cloudscape database, which is otherwise stored as clear text. You must specify a boot password when you boot the database in order to connect to it.

**Durability**

The D in ACID. Durability means that a transaction’s committed updates must be durable, even in the case of hardware or software failure.

**Durable transaction**

In Cloudscape synchronization, a transaction that has committed at the source.

**Dynamic parameter**

A parameter to a `PreparedStatement` whose value is unspecified when the statement is prepared. A dynamic parameter is represented by a question mark (?) in the SQL-J language.

**Embedded environment**

When an application starts up an instance of Cloudscape within its JVM, the application is said to run in an embedded environment. In this environment, only a single application can connect to a database at one time, and no network access occurs. The end-user may not even be aware of the database.

**Environment**

The way your application interacts with Cloudscape. Sometimes referred to as a framework. The two environments are Embedded environment and Client/server environment.

**External virtual table**

A virtual table in a FROM clause of an SQL-J statement in which a constructor of a class that fulfills Cloudscape’s Virtual Table Interface (VTI) requirements makes external data available to a SELECT statement.

**Field access**

In Cloudscape, accessing the static field of a Java class, or accessing the static or non-static field of an instance of a class. The class does not need to be serializable or stored in the database.
Hash join
A join operation in which Cloudscape uses a hash table for performance reasons. It is suitable for joins in which the outer table’s values are joined with single rows in the inner table, and values in the inner table are unique. See Nested loop join for comparison.

Index
A dictionary object that helps the DBMS find rows of a table quickly based on the values of one or more columns.

Inner table
In a join operation, a table that is not the outermost table. See Outer table for more information.

Isolation
The I in ACID. Isolation means that the database must give each connection data that appear to be unaffected by other transactions until the transaction commits.

Java data type
A Java class used within a Cloudscape system for storing instances in a table or for method invocation, or both. Also known as a user-defined data type. The class must be available to the Cloudscape system.

Java stored procedure
You may hear from other vendors about stored procedures written in Java that are declared with a statement such as CREATE PROCEDURE. These stored procedures are called Java stored procedures. In Cloudscape, you can call any Java method without having to declare it. In this documentation, we call such methods database-side methods, because you have the DBMS execute them. Such methods do not have to be stored in the database.

JDBC
The java.sql package, which is a set of classes and interfaces that make it possible to access databases from a Java application. (JDBC is not an acronym.)

JDBC driver
A Java class that implements the JDBC driver interface and is loaded into the JDBC driver manager. The core Cloudscape product comes with a built-in, local JDBC driver.

JDBC method
A method in a Java class that uses JDBC calls to access data in a Cloudscape database. When invoked from within Cloudscape, the method is called a database-side JDBC method.

JDK
Java Development Kit. The software released by JavaSoft to support the Java environments. The JDK includes the Java compiler and interpreter and many Java libraries.

JIT
Just-in-Time compiler. An alternative to a JVM; instead of interpreting byte code, it compiles it into machine code on the fly.

Join
The basic relational operator that allows data from more than one table to be combined.

JRE
Java Runtime Environment. Consists of the Java virtual machine, the Java platform core classes, and supporting files. It is the runtime part of the Java Development Kit—no compiler,
no debugger, no tools. The JRE is the smallest set of executables and files that constitute the standard Java platform. A JRE is a JVM targeted primarily for developers who will be bundling the JRE with their applications for deployment.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM (Java Virtual Machine)</td>
<td>The program that interprets the byte codes of a Java application. A JVM is part of a JDK.</td>
</tr>
<tr>
<td>Keyword</td>
<td>A word that is part of a DBMS’s command vocabulary. The SQL-92 standard has defined a large number of keywords. Some are reserved, which means that they cannot be used for another purpose within an SQL statement unless delimited, and some aren’t. Cloudscape has its own additional keywords.</td>
</tr>
<tr>
<td>Latency</td>
<td>In a synchronized system, the time between a change at one database and the corresponding change at another database.</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol. It provides an open directory access protocol running over TCP/IP. You can configure Cloudscape to authenticate users through an LDAP Directory Server.</td>
</tr>
<tr>
<td>Leaf level</td>
<td>The bottom level of a BTREE index.</td>
</tr>
<tr>
<td>Long row</td>
<td>A row that spans more than one database page. Cloudscape automatically overflows long rows that do not fit on a single database page.</td>
</tr>
<tr>
<td>LUCID</td>
<td>Logic up, consistent information down. This acronym describes Cloudscape’s synchronization model.</td>
</tr>
<tr>
<td>Metadata</td>
<td>Information about a database or a DBMS. Examples: the kind of functionality a DBMS provides, the types of data in the database, or information about a result set. In JDBC, you call the methods of the <code>DatabaseMetaData</code> or the <code>ResultSetMetaData</code> class to get this information.</td>
</tr>
<tr>
<td>Method</td>
<td>A procedure associated with a class or interface that defines one of the legal operations on instances of the class or interface. In Cloudscape, you can invoke methods that belong to a class stored in the database or to any class visible to Cloudscape.</td>
</tr>
<tr>
<td>Method Invocation</td>
<td>In Cloudscape, invoking a Java class method. The method can be either static or not static.</td>
</tr>
<tr>
<td>Nested connection</td>
<td>The “current” connection “re-used” by a database-side JDBC method. Such methods re-use the current connection so that statements that they execute are contained by a single transaction.</td>
</tr>
<tr>
<td>Nested loop join</td>
<td>The typical way that Cloudscape executes joins. Cloudscape loops through the inner table using the best access method available (usually an index) for every qualifying row in the outer table.</td>
</tr>
</tbody>
</table>
Cloudscape Glossary

Null
A data value that is unknown or unspecified. Java has a distinguished null object value, and SQL has a distinguished null value that represents the absence of an actual value.

Object
An instance of a Java class. An object can be stored in a Cloudscape database if it implements the java.io.Serializable interface.

Optimization
A process in which the DBMS makes the best (optimal) choice of access paths and join order. True query optimization means that the DBMS will usually make the best choice independent of how the query is written. Cloudscape’s optimizer usually makes a good decision; you may sometimes want to override it.

Orderable Java data type
In Cloudscape, Java data types that can be used in ordering operations such as ORDER BY, GROUP BY, DISTINCT, UNION and > and < comparisons. You can also create indexes on orderable Java data types.

Outer table
In a join operation, the table that Cloudscape looks at first. It is best if the outer table is the one for which the fewest rows qualify, since Cloudscape will look up values in an inner table or tables for every qualifying row in the outer table.

Package
In a Java library of classes, a group of classes and interfaces. Package names are separated by dots and stored in directories that match these names. When specifying Java classes within an SQL-J statement, you must specify the package name along with the class name.

Page
A user-configurable unit of storage within a conglomerate. Conglomerates are made up of multiple pages.

Positioned update/delete
A mechanism to update or delete the row at the current position through an SQL cursor.

Prepared statement
An SQL statement that has been compiled for efficiency. The SQL processor parses and analyzes the statement and produces an execution plan for the prepared statement.

Primary key
The columns in a table that uniquely identify the rows in the table.

Property
In the Java world, a configuration parameter. You can set properties for Cloudscape systems, databases, queries, or single conglomerates.

Provisional transaction
In Cloudscape synchronization, a transaction that has committed at a target but has not yet committed at the source.

Publication
In Cloudscape synchronization, a dictionary object in a source database that defines a set of target databases. Source databases have publications; target databases subscribe to publications.

Pure Java
Refers to a program written entirely in Java. The tag “100% Pure” carries additional restrictions.

Query
The basic part of a SELECT statement.
RDBMS  Relational database management system. A DBMS that supports the SQL standard is considered relational.

Refresh  In a Cloudscape synchronization, a request issued by a target database to submit changes to the source and copy source changes to the target. All synchronization in Cloudscape is done using the refresh operation.

Result set  The rows that satisfy the conditions of a query. Cloudscape returns these rows through the JDBC interface in a JDBC ResultSet object. You can access the result set one row at a time.

Rollback  Undoing the changes made by a transaction before it commits. You can initiate a rollback by explicitly calling the JDBC Connection method rollback.

Row-Level locking  A locking system in which Cloudscape locks only rows, not entire tables. Row-level locking allows greater concurrency. Cloudscape uses locking to present data in a consistent state to multiple users or applications.

Schema  (1) A description of the dictionary objects in a database and their attributes.

  (2) A database entity that consists of one or more tables, views, triggers, and so on. A schema provides a way to group a subset of tables within a database. A schema name qualifies a table name within a database.

Serialization  A Java mechanism for writing objects to an output stream. Cloudscape uses the serialization mechanism to store objects in a database.

Server  A machine that provides services of some kind to client machines on the network. A database server is a DBMS that provides services to applications that reside on client machines on the network. Cloudscape can be deployed as a database server by using Cloudconnector or another server framework.

Server framework  An application that enables multiple connections to Cloudscape. Unless Cloudscape runs inside a server framework, only one application at a time can access a Cloudscape database. Cloudconnector and RmiJdbc are server frameworks.

Source  In a Cloudscape synchronization system, the database from which data are copied to other databases (called targets).

SQL  A standardized database language for specifying queries and updates to databases. (At one time, SQL stood for Structured Query Language; now it’s another one of those acronyms, like JDBC, that do not stand for anything.)

SQL-J  Cloudscape’s Java-extended dialect of SQL-92.

SQL-92 entry-level

A subset of full SQL-92 specified by ANSI/ISO that is supported by nearly all major DBMSs today. JDBC compliance requires entry-level SQL compliance and defines escape syntax and metadata routines for more advanced features.

SQLExceptions

Errors generated during database access.

SQLState

Standard error code, defined by SQL-92, associated with SQLExceptions.

SSL

Secure Socket Layer. A protocol standard in wide use today for negotiating a secure connection.

Static method

A method that is associated with a class rather than an instance of the class.

Stored procedure

In some DBMSs, procedural code, written in SQL, executed by the DBMS. In Cloudscape, you execute Java methods instead of stored procedures.

Stored prepared statement

SQL-J statements named by the user and stored in a precompiled state for performance reasons.

Streaming column

Data stored via an input stream or retrieved via an output stream.

Synchronization

An operation that keeps distributed databases in agreement.

System

Semi-persistent environment for a Cloudscape session. Also a location; the default location for Cloudscape databases and configuration properties.

Target

In a Cloudscape synchronization system, a database created from a publication. See Publication.

Tear-Off Database

A target database.

Transaction

A sequence of JDBC/SQL calls within a single connection that constitute an atomic unit of work, that provide atomicity, consistency, integrity of data, and durability of database changes. In Cloudscape, transactions are handled entirely by JDBC calls.

URL

Uniform resource locator. First used on the WWW to identify HTML documents. A specialized form of URL is used by JDBC to identify databases. See Database connection URL.

User-defined data type

See Java data type.

Virtual Table Interface (VTI)

A Cloudscape construct that provides an interface to external data within an SQL-J statement. Classes that fulfill the VTI requirements can be used in the FROM clause to provide data to a SELECT statement. Cloudscape’s bulk import utility, COM.cloudscape.vti.FileImport, is an example of such a class.

WHERE clause

The portion of an SQL statement that allows the statement to retrieve a subset of rows that meet a particular set of conditions. Retrieving a subset of rows is sometimes called restriction. The WHERE clause requires a search condition (one or more predicates combined with AND, OR, and NOT) to specify the con-
ditions under which you want rows to be chosen for inclusion in your set of data.
In a Cloudscape synchronization system, a table in a publication can include a WHERE clause.

Two static Java methods, one at the target and one at the source, that perform a single logical operation and are referred to by a single alias.
Index

A
Access path 7-17
ACID properties of transactions 9-2
glossary definition G-1
Aggregates 8-11–8-15, 9-14
Agreement
glossary definition G-1
Altering classes 10-5
Applets
common problems 11-15
degrossary definition G-1
using client driver 11-12
Application logic
classification-side vs. database-side execution
6-3
Application servers 5-3
Applications
developing 5-6–5-9
Atomicity
glossary definition G-1
Auto-commit 6-12
glossary definition G-1
AVG aggregate function 8-12

B
Backing indexes
glossary definition G-1
Binding stage of compilation 7-2
Bulk export
glossary definition G-1
Bulk import 8-5–8-7
glossary definition G-1

C
CALL statement
development of 6-20
CAST function
deglassary definition G-2
example use of 10-9
Check constraints
glossary definition G-1
Class compatibility 10-5
Class path 2-2–2-6
glossary definition G-1
testing 2-11, 11-2
Classes
toering 10-5
storing in the database 10-12–10-15
ClassNotFound exceptions
troubleshooting A-4
Cloudscape as server 5-3
cloudscape.properties file 5-11
cloudscape.system.home
toificance of setting 2-15
CLOUDSCAPE_INSTALL environment
toariable 2-3
Cloudview
to how use 3-3
Code generation
tage of compilation 7-2
Commit 6-12, 9-1
glossary definition G-1
Compilation
toediction 7-2
COM.cloudscape.tools.sysinfo -cp 2-11
Concurrency
glossary definition G-2
Conglomerates
toediction 7-12
Connecting to databases
to application 2-13
Connection object
to parameter to JDBC method 6-11
Connections
glossary definition G-2
Learning Cloudscape
Index

Cloudscape Version 3.0

extensions to Cloudscape 1-5
implications for deployment 1-7
Java constructors
  use of in SQL-J statements 6-18
Java data types
  glossary definition G-4
  storing 10-2
Java stored procedures
  glossary definition G-4
JBMS Tours sample application 1-8—1-15
  in client/server mode 1-14
  JBMS Tours.BuildATour 5-14
  JBMS Tours.GenerateReport 8-15
  JBMS Tours.serializabletypes.City 6-3
JDBC 5-6—5-9
  glossary definition G-4
  tutorial for 5-7
JDBC driver
  glossary definition G-4
JDBC methods
  glossary definition G-4
JDBC objects
  explicitly closing 5-13
JDBC-ODBC bridge 5-3
JDK
  glossary definition G-4
JIT
  glossary definition G-4
Join
  definition 3-2
Join order 7-17
Joins
  definition 4-6
  example 4-10
  glossary definition G-4
JRE (Java runtime environment)
  glossary definition G-4
JVMs (Java Virtual Machines)
  glossary definition G-5
K
Keys
  definition 3-2
Keywords
  glossary definition G-5
L
Latency
  glossary definition G-5
LDAP
  glossary definition G-5
Leaf level
  glossary definition G-5
Locks 9-3
  exclusive 9-4, 9-8, 9-10
  shared 9-3, 9-7
Long rows
  glossary definition G-5, G-5
LUCID
  glossary definition G-5
M
Matching index scans
  definition 7-14
MAX aggregate function
  example 8-12
Metadata
  glossary definition G-5
Method aliases 6-18
Method invocation
  glossary definition G-5
  overview 1-6
Methods
  executing from Cloudview 4-3
  executing within SQL-J statements 4-8
  glossary definition G-5
  requirements for invoking 6-14
MIN aggregate function
  example 8-12
N
Nested connections
  glossary definition G-5
Nested loop joins
  glossary definition G-5
NEW (Java constructor) 6-18
Not enough environment space
  troubleshooting A-2
Null
  object definition G-6
O
Objects
  browsing in Cloudview 4-3
  definition G-6
  executing methods of 6-6
  stored 4-2
  storing in the database 10-2—10-7
Optimization 7-12—7-18
  glossary definition G-6
  stage of compilation 7-2
ORDBMS
  definition 1-4
Outer tables
Packages glossary definition G-6
Pages glossary definition G-6
Parsing stage of compilation 7-2
Positioned update/deletes glossary definition G-6
Prepared statements 7-2—7-8
  glossary definition G-6
  in a Java program 7-6, 7-8, 7-12
  in ij 7-4
  performance benefits of 7-3
  using 5-7
Primary keys glossary definition G-6
Properties 5-11
  glossary definition G-6
  sample use of 5-11
Provisional transactions glossary definition G-6
Publications glossary definition G-6
Queries definition 4-5
RDMBS (relational database management system)
definition 1-3
  glossary definition G-7
Refresh glossary definition G-7
Replication overview 1-7
Result sets glossary definition G-7
ResultSets working with 5-7
RmiJdbc server starting 11-2
  stopping 11-5
Rollback 6-12, 9-1, 9-10
  glossary definition G-7
Row-level locking benefits of 9-4
glossary definitions G-7
RunTimeStatistics 7-17
Schematics glossary definition G-7
Serialization glossary definition G-7
serialVersionUID 10-6
Server frameworks 11-2
Servers glossary definition G-7
Servlets 11-5
  configuring 11-6
  examples 11-11
Shutdown function 3-11
Shutting down Cloudscape from an application 2-15
Sorting results ORDER BY 6-8
Sources glossary definition G-7
SQL case-insensitivity of 4-6
definition 1-3
  uses 4-5
SQLExceptions glossary definition G-8
  handling 5-9
SQL-92 definition 1-3
Starting Cloudscape from an application 2-10
Statements using 5-7
  when they are optimizable 7-13
Static methods glossary definition G-8
  invoking 6-15
Stored prepared statements 7-8—7-11
  differences from views 8-3
  glossary definition G-8
Stored procedures 6-19
  glossary definition G-8
  invoking 6-16
Storing classes in the database 10-12—10-15
Streaming columns glossary definition G-8
Strings delimited by single quotation marks in SQL-J 3-8
Subclasses
accessing methods of 10-8
storing 10-8
Synchronization
glossary definition G-8
System
glossary definition G-8
System directory 2-6–2-7, 3-4

T
Table scan
forcing 7-15
Table-level locking 9-4
Tables
definition 3-2
Targets
glossary definition G-8
Technology overview 1-1
toursDB sample database 1-8–1-15
how to build 2-17
installed with product 2-17
Transaction control 1-3
Transactions 9-1–9-11
ACID properties 9-2
glossary definition G-8
TRANSACTION_SERIALIZABLE isolation level 9-3

U
URL
glossary definition G-8

V
VALUES clause 6-13
Views 8-3
differences from stored prepared statements 8-3
Virtual Table Interface (VTI)
glossary definition G-8
Virtual tables 8-2

W
WHERE clause
definition 4-6
glossary definition G-8
Work units
glossary definition G-9

Symbols
? (dynamic parameter) 7-4