Tuning Cloudscape

Cloudscape Version 3.0
December 15, 1999

Table of Contents

About This Document *xi*

Purpose of This Document *xi*

Audience *xi*

How This Document Is Organized *xii*

1 Working with Cloudscape Properties *1-1*

Properties Overview *1-1*

- Scope of Properties *1-2*
- Persistence of Properties *1-2*
- Precedence of Properties *1-3*
  - Protection of Database-Wide Properties *1-3*
- Dynamic vs. Static Properties *1-4*

Ways of Setting Cloudscape Properties *1-4*

- System-Wide Properties *1-4*
  - Programmatically *1-4*
    - As a Parameter to the JVM Command Line *1-5*
    - Using a Properties Object Within an Application or Statement *1-5*
  - In the *cloudscape.properties* File *1-6*
  - Verifying System Properties *1-6*
- Database-Wide Properties *1-7*
  - Shortcut for Setting Database Properties *1-8*
- Conglomerate-Specific Properties *1-8*
  - Verifying Conglomerate-Specific Properties *1-9*
- In a Client/Server Environment *1-10*

Dynamic or Static Changes to Properties 1-10
Properties Case Study 1-11

2 Performance Tips and Tricks 2-1
The Top Ten Tips 2-1
Tip One. Use a Fast JVM with Lots of Memory, and Tune the JIT 2-2
Tip Two. Use Stored Prepared Statements and Prepared Statements 2-3
Tip Three. Create Indexes, and Make Sure They're Being Used 2-4
Tip Four. Tune How You Load Data 2-4
Tip Five. Increase the Size of the Data Page Cache 2-4
Tip Six. Tune the Size of Database Pages 2-5
Performance Trade-Offs of Large Pages 2-6
When Large Page Size Does Not Improve Performance 2-6
When Large Page Size Is Not Desirable 2-6
Tip Seven. Avoid Expensive Queries 2-7
Tip Eight. Use the Appropriate *getXXX* and *setXXX* Methods for the Type 2-7
Tip Nine. Tune Database Booting/Class Loading 2-8
Tip Ten. Recompile Stale Stored Prepared Statements 2-9
No Longer in the Top Ten, but Worth Listing 2-10
Shut Down the System Properly 2-10
Shorten Your Class Path 2-10
Increase the Statement Cache Size 2-10
When Working with Development Tools, Pre-Load DatabaseMetaData Stored Prepared Statements 2-11

3 Tuning Databases and Applications 3-1
Application and Database Design Issues 3-1
Avoiding Table Scans of Large Tables 3-1
Index, Index, Index 3-2
Create Useful Indexes 3-2
Make Sure They Are Being Used, and Rebuild Them 3-3
Think About Join Order 3-3
Prevent the User from Issuing Expensive Queries 3-4
Understand When Statements Go Stale 3-4

Overview *3-5*

Automatic Stale Plan Invalidation *3-5*

Stale Plans and Stored Prepared Statements *3-6*

When a Change in Table Makes a Plan *Stale* *3-7*

Avoiding Compiling SQL Statements *3-8*

Across Connections and Sessions *3-8*

Within a Single Connection *3-8*

Shielding Users from Cloudscape Class-Loading Events *3-10*

Tuning Tips for Multi-User Systems *3-11*

Analyzing Statement Execution *3-12*

Tuning Tips for Single-User Systems *3-12*

Storing Prepared Statements to Improve Performance *3-13*

Creating a Stored Prepared Statement *3-13*

Executing a Stored Prepared Statement *3-14*

Additional Benefits for Multiple Connections: The Stored Prepared Statement Cache *3-15*

Invalid Stored Prepared Statements and Recompilation *3-17*

Stale Stored Prepared Statements *3-18*

Working with RunTimeStatistics *3-18*

Overview *3-19*

How You Use It *3-19*

Analyzing the Information *3-20*

Statistics Timing *3-20*

Statement Execution Plan *3-21*

Subset of Statement Execution Plan *3-23*

Optimizer Estimates *3-24*

Viewing Runtime Statistics in Cloudview *3-24*

4 DML Statements and Performance *4-1*

Performance and Optimization *4-1*

Index Use and Access Paths *4-2*

What Is an Index? *4-2*

What's Optimizable? *4-5*

Directly Optimizable Predicates *4-5*

Indirectly Optimizable Predicates *4-6*

Joins *4-7*

Covering Indexes *4-7*

Single-Column Index Examples *4-7*

Multiple-Column Index Example *4-8*

Useful Indexes Can Use Qualifiers *4-9*

When a Table Scan Is Better *4-10*

Indexes Have a Cost for Inserts, Updates, and Deletes *4-10*

How Indexes Affect Cursors *4-11*

Joins and Performance *4-11*

Join Order Overview *4-11*

Join Strategies *4-13*

Cloudscape's Cost-Based Optimization *4-14*

About the Optimizer's Choice of Access Path *4-14*

Estimating Row Counts for Unknown Search Values *4-15*

About the Optimizer's Choice of Join Order *4-16*

Join Order Case Study *4-17*

About the Optimizer's Choice of Join Strategy *4-18*

About the Optimizer's Choice of Sort Avoidance *4-18*

Cost-Based ORDER BY Sort Avoidance *4-19*

About the Optimizer's Selection of Lock Granularity *4-21*

How the Optimizer Makes Its Decision at Compile Time *4-22*

Lock Escalation Threshold *4-24*

Runtime Overrides *4-24*

About the Optimizer's Selection of Bulk Fetch *4-24*

Overriding the Optimizer *4-25*

Overriding the Optimizer's Choice of Access Path *4-25*

Overriding the Optimizer's Choice of Join Order *4-25*

Overriding the Optimizer's Choice of Join Strategy *4-26*

Overriding the Optimizer's Choice of Fetch Size *4-26*

Tuning Lock Granularity *4-26*

Optimizer Accuracy *4-27*

Providing Costing Information for VTIs *4-27*

Locking and Performance *4-28*

Transaction-Based Lock Escalation *4-28*
LOCK TABLE Statement *4-30*
Non-Cost-Based Optimizations *4-30*
Non-Cost-Based Sort Avoidance (Tuple Filtering) *4-30*
DISTINCT *4-31*
Quick DISTINCT Scans *4-31*
GROUP BY *4-32*
The MIN() and MAX() Optimizations *4-32*

5 Cloudscape Properties *5-1*

cloudscape.authentication.ldap.searchAuthDN *5-5*
cloudscape.authentication.ldap.searchAuthPW *5-7*
cloudscape.authentication.ldap.searchBase *5-8*
cloudscape.authentication.ldap.searchFilter *5-9*
cloudscape.authentication.provider *5-11*
cloudscape.authentication.server *5-13*
cloudscape.connection.requireAuthentication *5-15*
cloudscape.database.classpath *5-17*
cloudscape.database.defaultConnectionMode *5-19*
cloudscape.database.forceDatabaseLock *5-21*
cloudscape.database.fullAccessUsers *5-22*
cloudscape.database.noAutoBoot *5-24*
cloudscape.database.propertiesOnly *5-25*
cloudscape.database.readOnlyAccessUsers *5-26*
cloudscape.infolog.append *5-28*
cloudscape.jdbc.metadataStoredPreparedStatements *5-29*
cloudscape.language.bulkFetchDefault *5-31*
cloudscape.language.defaultIsolationLevel *5-33*
cloudscape.language.logStatementText *5-35*
cloudscape.language.preloadClasses *5-36*
cloudscape.language.spsCacheSize *5-37*
cloudscape.language.stalePlanCheckInterval *5-39*
cloudscape.language.statementCacheSize *5-41*
cloudscape.language.triggerMaximumRecursionLevel *5-43*
cloudscape.locks.deadlockTimeout *5-45*
cloudscape.locks.deadlockTrace *5-47*
cloudscape.locks.escalationThreshold *5-48*
cloudscape.locks.monitor *5-50*
cloudscape.locks.waitTimeout *5-51*
cloudscape.service *5-53*
cloudscape.storage.initialPages *5-55*
cloudscape.storage.minimumRecordSize *5-57*
cloudscape.storage.pageCacheSize *5-59*
cloudscape.storage.pageReservedSpace *5-60*

cloudscape.storage.pageSize *5-62*
cloudscape.storage.rowLocking *5-64*
cloudscape.storage.tempDirectory *5-66*
cloudscape.stream.error.field *5-68*
cloudscape.stream.error.file *5-69*
cloudscape.stream.error.logSeverityLevel *5-70*
cloudscape.stream.error.method *5-71*
cloudscape.system.bootAll *5-72*
cloudscape.system.home *5-73*
cloudscape.user.*UserName* *5-74*

6 Optimizer Overrides *6-1*

bulkFetch *6-3*
constraint *6-5*
index *6-8*
joinOrder *6-9*
joinStrategy *6-10*

Appendix A
Internal Language Transformations *A-1*

Predicate Transformations *A-2*
BETWEEN Transformations *A-3*
LIKE Transformations *A-3*
Character String Beginning with Constant *A-4*
Character String Without Wildcards *A-4*
Unknown Parameter *A-5*
Static IN Predicate Transformations *A-5*
NOT IN Predicate Transformations *A-6*
OR Transformations *A-6*
Transitive Closure *A-7*
Transitive Closure on Join Clauses *A-7*
Transitive Closure on Search Clauses *A-7*
View Transformations *A-9*
View Flattening *A-9*
Predicates Pushed into Views or Derived Tables *A-10*
Subquery Processing and Transformations *A-11*
Materialization *A-11*
Flattening a Subquery into a Normal Join *A-12*

Flattening a Subquery into an EXISTS Join *A-15*
DISTINCT Elimination in IN, ANY, and EXISTS Subqueries *A-16*
IN/ANY Subquery Transformation *A-16*
Sort Avoidance *A-17*
DISTINCT Elimination Based on a Uniqueness Condition *A-17*
Combining ORDER BY and DISTINCT *A-19*
Combining ORDER BY and UNION *A-20*
Aggregate Processing *A-21*
COUNT(nonNullableColumn) *A-21*

Index *IN-1*

About This Document

- "Purpose of This Document" on page xi
- "Audience" on page xi
- "How This Document Is Organized" on page xii

For general information about the Cloudscape documentation, such as a complete list of books, conventions, and further reading, see *Using the Cloudscape Documentation*.

Purpose of This Document

This book, *Tuning Cloudscape*, explains how to set properties in Cloudscape, which is how you configure and tune systems, databases, specific tables and indexes, and queries. It also provides performance tuning tips and an in-depth study of query optimization and performance issues.

Audience

This book is a reference for Cloudscape users, typically application developers. Cloudscape users who are not familiar with the SQL standard or the Java™ programming language will benefit from consulting books on those topics.

Cloudscape users who want a how-to approach to working with Cloudscape or an introduction to Cloudscape concepts should read the *Cloudscape Developer's Guide*. This book is for users who want to optimize and tune their application's performance.

How This Document Is Organized

This document includes the following chapters:

- *Chapter 1, "Working with Cloudscape Properties"*
  An overview of how you set properties.
- *Chapter 2, "Performance Tips and Tricks"*
  Quick tips on how to improve the performance of Cloudscape applications.
- *Chapter 3, "Tuning Databases and Applications"*
  A more in-depth discussion of how to improve the performance of Cloudscape applications.
- *Chapter 4, "DML Statements and Performance"*
  An in-depth study of how Cloudscape executes queries, how the optimizer works, and how to tune query execution.
- *Chapter 5, "Cloudscape Properties"*
  Reference on Cloudscape properties.
- *Chapter 6, "Optimizer Overrides"*
  Reference on special properties that override the optimizer's decisions about the execution of a specific query.
- *Appendix A, "Internal Language Transformations"*
  Reference on how Cloudscape internally transforms some SQL-J statements for performance reasons. Not of interest to the general user.

1 Working with Cloudscape Properties

This chapter includes the following sections:

- "Properties Overview" on page 1-1
- "Ways of Setting Cloudscape Properties" on page 1-4
- "Properties Case Study" on page 1-11

Properties Overview

Cloudscape lets you configure behavior or attributes of a system, a specific database, or a specific *conglomerate* (a table or index) through the use of properties.

Examples of behavior or attributes that you can configure are:

- Whether to authorize users
- Page size of tables and indexes
- Where and whether to create an error log
- Which databases in the system to boot

In addition, you may sometimes be able to use properties to affect something as specific as the access path for the execution of a single query. See "Overriding the Optimizer" on page 4-25 for a discussion of how these properties are used. This chapter does not discuss that type of property, which you set in a different way.

For reference information about specific properties, see Chapter 5, "Cloudscape Properties".

Scope of Properties

You use properties to configure a Cloudscape system, database, or conglomerate.

- *system-wide*
  Most properties can be set on a *system-wide* basis; that is, you set a property for the entire system and all its databases and conglomerates, if this is applicable. Some properties, such as error handling and automatic booting, can be configured only in this way, since they apply to the entire system. (For information about the Cloudscape system, see "Cloudscape System" on page 2-5 of the *Cloudscape Developer's Guide*.)

  When you change these properties, they affect any tables or indexes created *after* this change.
- *database-wide*
  Some properties can also be set on a *database-wide* basis. That is, the property is true for the selected database only and not for the other databases in the system unless it is set individually within each of them. In addition, in order to publish properties to target databases in a distributed system, you publish them as database-wide properties.

  When you change these properties, they affect any tables or indexes created *after* this change.
- *conglomerate-specific*
  Finally, some properties can also be set on a *conglomerate-specific* basis. When set this way, these properties are true for the selected conglomerate only, and not for the other conglomerates in the database or system. These storage-related properties take effect at table or index creation, and cannot be changed during the lifetime of the conglomerate. (The properties of a table at a source are always automatically propagated to the target. For example, when you add a table to a publication, that table retains the properties of the table at the source, whether you have set those properties for the specific conglomerate or the table has inherited some database- or system-wide settings.)

Persistence of Properties

A database-wide property always has persistence. That is, its value is stored in the database. Typically, it is in effect until you explicitly change the property or until you set a system-wide property with precedence over database-wide properties (see "Precedence of Properties" on page 1-3).

A system-wide property may or may not have persistence, depending on how you set it. If you set it programmatically, it persists only for the duration of the JVM of the application that set it. If you set it in the *cloudscape.properties* file, a property persists until:

- that value is changed
- the file is removed from the system
- the database is booted outside of that system

Precedence of Properties

The search order for properties is:

1. Conglomerate-specific properties
2. System-wide properties set programmatically (as a command-line option to the JVM when starting the application or within application code)
3. Database-wide properties
4. System-wide properties set in the *cloudscape.properties* file

This means, for example, that storage properties set for a specific conglomerate override all system-wide and database-wide storage properties; that system-wide properties set programmatically override database-wide properties and system-wide properties set in the *cloudscape.properties* file, and that database-wide properties override system-wide properties set in the *cloudscape.properties* file.

Protection of Database-Wide Properties

There is one important exception to the search order for properties described above: When you set the *cloudscape.database.propertiesOnly* property to *true*, database-wide properties cannot be overridden by system-wide properties.

This property ensures that a database's environment cannot be modified by the environment in which it is booted. Typically, most databases that are distributed or synchronization targets require this property to be set to *true*. Many applications running in an embedded environment may set this property to true for security reasons.

Dynamic vs. Static Properties

Most properties are dynamic; that means you can set them while Cloudscape is running, and their values change without requiring a reboot of Cloudscape. In some cases, this change takes place immediately; in some cases, it takes place at the next connection.

Some properties are static, which means you cannot change their values while Cloudscape is running. You must restart or set them before (or while) starting Cloudscape.

For more information, see “Dynamic or Static Changes to Properties” on page 1-10.

Ways of Setting Cloudscape Properties

This section covers the different ways of setting properties.

- “System-Wide Properties” on page 1-4
- “Database-Wide Properties” on page 1-7
- “Conglomerate-Specific Properties” on page 1-8
- “In a Client/Server Environment” on page 1-10
- “Dynamic or Static Changes to Properties” on page 1-10

System-Wide Properties

You can set system-wide properties programmatically (as a command-line option to the JVM when starting the application or within application code) or in the text file *cloudscape.properties*.

Programmatically

You can set properties programmatically—either in application code before booting the Cloudscape driver or as a command-line option to the JVM when booting the application that starts up Cloudscape. When you set properties programmatically, these properties persist only for the duration of the application. Properties set programmatically are not written to the *cloudscape.properties* file or made persistent in any other way by Cloudscape.

NOTE: Setting properties programmatically works only for the application that starts up Cloudscape—for example, for an application in an embedded environment or for the application server that starts up a server product. It does not work for client applications connecting to a running server.

As a Parameter to the JVM Command Line

You can set system-wide properties as parameters to the JVM command line when starting up the application or framework in which Cloudscape is embedded.

- *JavaSoft and SunSoft JDKs*
  With the JavaSoft and SunSoft JDKs, you set JVM system properties using a *-D* flag on the Java command line. For example:

```
java -Dcloudscape.system.home=C:\home\jbms\
    -Dcloudscape.storage.pageSize=8192 JDBCTest
```

- *Microsoft SDK*
  With the Microsoft SDK, you set JVM system properties using a */d:* flag on the jview command line. For example:

```
jview /d:cloudscape.system.home=C:\home\jbms\ JDBCTest
```

For other JVMs, see the JVM-specific documentation on setting system properties.

Using a Properties Object Within an Application or Statement

In embedded mode, your application runs in the same JVM as Cloudscape, so you can also set system properties within an application using a *Properties* object before loading the Cloudscape JDBC driver. The following example sets *cloudscape.system.home* on Windows:

```
Properties p = System.getProperties();
p.put("cloudscape.system.home", "C:\databases\tours");
```

In client/server mode, you can accomplish the same thing within an SQL-J statement (this works in embedded mode, too):

```
CALL (CLASS java.lang.System).getProperties().put(
    'cloudscape.storage.pageSize', '1024')
```

NOTE: You cannot set any static system-wide properties (which require you to reboot Cloudscape) in an SQL-J statement. *cloudscape.system.home* is one of those properties.

NOTE: If you pass in a *Properties* object as an argument to the *DriverManager.getConnection* call when connecting to a database, those

properties are used as database connection URL attributes, not as properties of the type discussed in this book.

In the *cloudscape.properties* File

You can set persistent system-wide properties in a text file called *cloudscape.properties*, which lives in the directory specified by the *cloudscape.system.home* property. There should be one *cloudscape.properties* file per system, not per database. The file lives in the system directory. In a client/server environment, that directory is on the server. (For more information about a Cloudscape system and the system directory, see "Cloudscape System" on page 2-5 in the *Cloudscape Developer's Guide*.)

Cloudscape does *not*:

- provide this file
- automatically create this file for you
- automatically write any properties or values to this file

Instead, you must create, write, and edit this file yourself.

The file should be in the format created by the *java.tools.Properties.save* method. A sample *cloudscape.properties* files is provided in */demo/programs/tours/scripts*.

The following is the text of a sample properties file:

```
cloudscape.infolog.append=true

cloudscape.storage.pageSize=8192

cloudscape.storage.pageReservedSpace=60
```

Properties set this way are persistent for the system until changed, until the file is removed from the system, or until the system is booted in some other directory (in which case Cloudscape would be looking for *cloudscape.properties* in that new directory). If a database is removed from a system, system-wide properties do not "travel" with the database unless explicitly set again.

Verifying System Properties

You can find out the value of a system property if you set it programmatically. You cannot find out the value of a system property if you set it in the *cloudscape.properties* file.

For example, if you set the value of the system-wide property in an SQL-J statement, in a program, or on the command line, the following SQL-J statement returns its value:

```
VALUES (CLASS java.lang.System).getProperty(
    'cloudscape.storage.pageSize')
```

Database-Wide Properties

Database-wide properties, which affect a single database, are stored within the database itself. This allows different databases within a single Cloudscape system to have different properties and ensures that the properties are correctly set when a database is moved away from its original system or copied.

NOTE: Cloudscape recommends that you use database-wide properties wherever possible for ease of deployment.

You set and verify database-side properties using methods of the *COM.cloudscape.database.PropertyInfo* interface (aliased as *PropertyInfo*) within SQL-J statements. Use this interface only within the context of SQL-J statements. Cloudscape provides a class alias for this class, *PropertyInfo*.

NEW: The new alias, *PropertyInfo*, makes it easier to work with database-level properties.

To set a property, connect to the database, create a statement, and then use the *setDatabaseProperty* method, passing in the name and value of the property.

To check the current value of a property, connect to the database, create a statement, and then use the *getDatabaseProperty* method, passing in the name of the property. To check the current values of *all* database-wide properties, use the *getProperties()* method.

```
-- set the statement cache size to 40
CALL PropertyInfo.setDatabaseProperty(
   'cloudscape.language.statementCacheSize', '40')

-- set the default page size for all new tables in the database
CALL PropertyInfo.setDatabaseProperty(
   'cloudscape.storage.pageSize', '1024')

-- find out the current database-wide properties
-- (returns a Properties object)
VALUES PropertyInfo.getDatabaseProperties()
```

If you specify an invalid value, Cloudscape uses the default value for the property.

Since you set database-wide properties within SQL-J statements, you can set these properties in embedded or client/server mode.

In addition, in a Cloudscape synchronization system, you set properties for target databases by publishing database-level properties. (You cannot publish system-level properties, but you can set system-level properties individually on targets.)

For more information, see the *Cloudscape Synchronization Guide*.

See the API for *COM.cloudscape.database.PropertyInfo* for complete information on the static methods.

Shortcut for Setting Database Properties

You can create a method alias for *setDatabaseProperty*. Cloudscape provides a script in */demo/util/methodalias* that creates method aliases for the methods in the *COM.cloudscape.database.PropertyInfo* class. For more information, see */demo/util/methodalias/readme.html*.

After you run the script, you will be able to set the property like this:

```
CALL SETDATABASEPROPERTY(
    'cloudscape.database.readOnlyAccessUsers', 'fred,guest')
```

Conglomerate-Specific Properties

You can apply some storage-related properties to specific conglomerates (tables or indexes) within a database. These properties, which you set as part of the CREATE TABLE or CREATE INDEX statement at table or index creation time, define the way rows are stored only for the specific table or index, including backing indexes for constraints; the rest of the database uses the database or system defaults. Overriding the default with conglomerate-specific properties is useful when you have one table or index with storage needs different from those of the rest of the tables or indexes in a database. For more information, see “PROPERTIES clause” on page 1-95 in the *Cloudscape Reference Manual*.

For indexes, you can set only *cloudscape.storage.pageSize* and *cloudscape.storage.initialPages*. For tables, you can set *cloudscape.storage.initialPages*, *cloudscape.storage.minimumRecordSize*, *cloudscape.storage.pageSize*, or *cloudscape.storage.pageReservedSpace*.

For example, the following SQL-J statement creates a table with a page size of 2048:

```
CREATE TABLE NewTable (a INT, b VARCHAR(50))
    PROPERTIES cloudscape.storage.pageSize=2048
```

NOTE: If you specify an invalid property name or an invalid value, Cloudscape uses the default value.

Verifying Conglomerate-Specific Properties

To verify which properties were used for a specific table or index, use the static methods of *COM.cloudscape.database.PropertyInfo*.

The static methods *getTableProperties* and *getIndexProperties* return a *Properties* object equal to the properties used when creating that conglomerate. The methods take the name of the index or table as a string parameter. Unnamed constraints use system-generated names.

To find out the system-generated name of an index that backs a constraint, query the system tables. The following example shows how to find out the name of the backing index for the primary key constraint on the table *Flights*.

```
SELECT constraintname
FROM sys.sysconstraints JOIN sys.systables USING(tableid)
WHERE sys.systables.tablename = 'FLIGHTS'
AND sys.sysconstraints.type = 'P'
```

You have a lot of flexibility in the way you find out the information. For example, you can use the *toString* method to see the properties listed as text strings. In ij, you could find out the values of the properties used for creating a table like this:

```
VALUES PropertyInfo.getTableProperties(
    'APP', 'FLIGHTS').toString()
```

NOTE: The table name string is case-sensitive. Tables and indexes created as non-delimited identifiers must be specified in all caps.

Or you could find out the values for a particular property like this:

```
VALUES PropertyInfo.
getIndexProperties('APP',
'ORIGINDEX').getProperty('cloudscape.storage.pageSize')
```

See the API for *COM.cloudscape.database.PropertyInfo* for complete information on the static methods.

You can also use Cloudview to see properties for tables and indexes.

In a Client/Server Environment

In a client/server environment, you must set the system properties for the *server*'s system. That means when you are using the *cloudscape.properties* file, the file exists in the *server*'s *cloudscape.system.home* directory. Client applications can set database-wide and conglomerate-specific properties, since they are set via SQL-J statements. Client applications can set dynamic system-wide properties in an SQL-J statement, as shown in the example in "Using a Properties Object Within an Application or Statement" on page 1-5.

For more information, see the *Cloudscape Server and Administration Guide.*

Table 1-1 Summary of Ways to Set Properties

Type of Property	How You Set It
System-wide	In *cloudscape.properties* OR Programmatically (as a command-line option to the JVM when starting the application or within application code)
Database-wide	In an SQL-J statement For synchronized systems, you can publish database-wide properties to targets within a publication in the CREATE PUBLICATION or ALTER PUBLICATION command at the source. See the *Cloudscape Synchronization Guide.*
Conglomerate-specific	In the CREATE TABLE, ALTER TABLE ADD CONSTRAINT, or CREATE INDEX statement

Dynamic or Static Changes to Properties

NOTE: Properties set in the *cloudscape.properties* file and on the command line of the application that boots Cloudscape are *always* static, because Cloudscape reads this file and those parameters only at startup. Conglomerate-specific properties, on the other hand, are always dynamic.

Only properties set in the following ways have the potential to be dynamic:

- as database-wide properties
- as system-wide properties via a *Properties* object in the application in which the Cloudscape engine is embedded

See Chapter 5, "Cloudscape Properties", for information about specific properties.

Properties Case Study

Cloudscape allows you a lot of freedom in configuring your system. This freedom can be confusing if you do not understand how properties work. You also have the option of not setting any and instead using the Cloudscape defaults, which are tuned for a single-user embedded system.

Imagine the following scenario of an embedded environment:

Your system has a *cloudscape.properties* file, a text file that lives in the system directory, which you have created and named *system_directory.* Your databases also live in this directory. The properties file sets the following property:

- *cloudscape.storage.pageSize* = 8192

You start up your application, being sure to set the *cloudscape.system.home* property appropriately:

```
java -Dcloudscape.system.home=c:\system_directory MyApp
```

You then create a new table:

```
CREATE TABLE table1
    (a INT,
    b VARCHAR(10))
```

Cloudscape takes the page size of 8192 from the system-wide properties set in the *cloudscape.properties* file, since the property has not been set any other way.

You shut down and then restart your application, setting the value of *cloudscape.storage.pageSize* to 4096 programmatically, as a parameter to the JVM command line:

```
java -Dcloudscape.system.home=c:\system_directory
    -Dcloudscape.storage.pageSize=4096 myApp
```

You establish a connection to the database and set the value of the page size for all new tables to 1024 as a database-wide property:

```
CALL PropertyInfo.
    setDatabaseProperty(
    'cloudscape.storage.pageSize', '1024')
```

You then create two new tables, setting the page size for one of the specific tables in the CREATE TABLE statement:

```
CREATE TABLE table2
    (a INT,
```

```
    b VARCHAR(10))
PROPERTIES cloudscape.storage.pageSize=2048

CREATE TABLE table3
    (a INT,
    b VARCHAR(10))
```

In this scenario, Cloudscape uses the page size 2048 for the table *table2*, since conglomerate-specific properties always override system- and database-wide properties. Since no page size was set for *table3*, Cloudscape uses the system-wide property of 4096. The database-wide property of 1024 does not override a system-wide property set programmatically.

You shut down the application, then restart, this time forgetting to set the system-wide property programmatically (as a command-line option to the JVM):

```
java -Dcloudscape.system.home=c:\system_directory myApplication
```

You then create another table:

```
CREATE TABLE table4
    (a INT,
    b VARCHAR(10))
```

Cloudscape uses the persistent database-wide property of 1024 for this table, since the database-wide property set in the previous session is persistent and overrides the system-wide property set in the *cloudscape.properties* file.

What you have is a situation in which four different tables each get a different page size, even though the *cloudscape.properties* file remained constant.

Remove the *cloudscape.properties* file from the system or the database from its current location (forgetting to move the file with it), and you could get yet another value for a new table.

To avoid this situation, be consistent in the way you set properties.

2 Performance Tips and Tricks

This chapter lists tips for improving the performance of your Cloudscape application. For a more in-depth discussion of performance, see Chapter 3, "Tuning Databases and Applications".

- "The Top Ten Tips" on page 2-1
- "No Longer in the Top Ten, but Worth Listing" on page 2-10

The Top Ten Tips

1. *Tip One. Use a Fast JVM with Lots of Memory, and Tune the JIT*. Using a fast JVM can make a *2X performance improvement*.
2. *Tip Two. Use Stored Prepared Statements and Prepared Statements* to save on costly compilation time. Stored prepared statements let you avoid most compilation altogether, and some Cloudscape applications have seen *2–10X performance increases* as a result of using *PreparedStatements* instead of *Statements*.
3. *Tip Three. Create Indexes, and Make Sure They're Being Used*. Indexes speed up queries dramatically if the table is much larger than the number of rows retrieved.
4. *Tip Four. Tune How You Load Data*. Create indexes before, check constraints after.
5. *Tip Five. Increase the Size of the Data Page Cache* and prime all the caches.

6. *Tip Six. Tune the Size of Database Pages*. Using large database pages has provided a performance improvement of *up to 50%*. There are also other storage parameters worth tweaking. If you use large database pages, increase the amount of memory available to Cloudscape.
7. *Tip Seven. Avoid Expensive Queries*.
8. *Tip Eight. Use the Appropriate getXXX and setXXX Methods for the Type*.
9. *Tip Nine. Tune Database Booting/Class Loading*. System startup time can be improved by reducing the number of databases in the system directory.
10. *Tip Ten. Recompile Stale Stored Prepared Statements if data distribution changed*. If you created a stored prepared statement when the table was empty, and now it has a lot of rows, you may need to recompile to get a better query plan.

These tips may or may not solve your particular performance problem. Be sure to visit the Support section of Cloudscape's Web site for up-to-date performance tips and tricks.

Tip One. Use a Fast JVM with Lots of Memory, and Tune the JIT

Not all Java Virtual Machines (JVMs) are created equal. Using a JVM with a Just-in-Time Compiler (JIT) can bring a 2X performance improvement to Cloudscape applications. Many JVMs are available for different hardware and operating system combinations. Here are some:

- *Javasoft JDK/JRE Performance Packs (available for Windows only)*
  The "vanilla" JDK or Java Runtime Environment (JRE) always includes a JIT. See the Javasoft JDK page for details.
- *Visual Cafe (available for Windows only)*
  If you're using Symantec Cafe, the JVM included with it has a JIT. See the Symantec Cafe home page.
- *The Microsoft Virtual Machine (available for Windows only)*
  The Microsoft Virtual Machine, also known as jview, includes a JIT compiler.
- *IBM Developer Kit and Runtime Environment for Windows*
  IBM's new JVM and JRE for Windows.

However, also try your application with and without a JIT. Sometimes GUI applications run faster without the JIT. Another reason to use -nojit is if startup time is more important than long-term performance.

For example: (JDK 1.1)

```
java -nojit COM.cloudscape.tools.cview
```

(JDK 1.2)

```
java -Djava.compiler=NONE COM.cloudscape.tools.cview
```

In addition, allocate as much memory as you can to the JVM to use. Use the minimum size flag upon start up; it can reduce the boot time by at least a couple of seconds. For example:

```
java -ms16m -mx16m COM.cloudscape.tools.ij
```

In a multi-user environment, use a machine with a lot of memory and allocate as much memory as you can for the JVM to use. Cloudscape can run in a small amount of memory, but the more memory you give it, the faster it runs.

NOTE: If you do not allocate memory using -mx, the JVM typically limits itself to 32 MB.

Tip Two. Use Stored Prepared Statements and Prepared Statements

You can name and store prepared statements in a database. Storing prepared statements helps you avoid costly compilation time, because the statement is stored precompiled. In your application, you do have to "prepare" a statement that references the stored prepared statement, but this statement has a minimal compilation cost.

You will find a more complete description of stored prepared statements in "Storing Prepared Statements to Improve Performance" on page 3-13.

Where stored prepared statements are not possible, use prepared statements. In Cloudscape, as with most relational database management systems, performing an SQL request has two steps: compiling the request and executing it. By using prepared statements (*java.sql.PreparedStatement*) instead of statements (*java.sql.Statement*) you can help Cloudscape avoid unnecessary compilation, which saves time. In general, any query that you will use more than once should be a stored prepared statement or prepared statement.

For more information, see "Avoiding Compiling SQL Statements" on page 3-8.

Making this change has resulted in 2–10X performance improvement, depending on the complexity of the query. More complex queries show more benefit from being prepared.

Tip Three. Create Indexes, and Make Sure They're Being Used

By creating indexes on columns by which you often search a table, you can reduce the number of rows that Cloudscape has to scan, thus improving performance. Depending on the size of the table and the number of rows returned, the improvement can be dramatic. Indexes work best when the number of rows returned from the query is a fraction of the number of rows in the table.

There are some trade-offs in using indexes: indexes speed up searches but slow down inserts and updates. As a general rule, every table should have at least a primary key constraint.

See "Index, Index, Index" on page 3-2 for more information.

Tip Four. Tune How You Load Data

When loading large amounts of data into tables, create indexes (including those created to back primary, unique, and foreign key constraints) before you load and check constraints after. Having indexes already defined before you bulk load is faster than creating the indexes after the data is loaded. Disable check constraints before loading, and re-enable after loading.

Tip Five. Increase the Size of the Data Page Cache

You can increase the size of a database's data page cache, which consists of the data pages kept in memory. When Cloudscape can access a database page from the cache instead of reading it from disk, it can return data much more quickly.

The default size of the data page cache is 40 pages. In a multi-user environment, or in an environment where the user accesses a lot of data, increase the size of the cache. You configure its size with the *cloudscape.storage.pageCacheSize* property. For more information about how to set this property and how to estimate memory use, see *"cloudscape.storage.pageCacheSize" on page 5-59.*

NOTE: Cloudscape can run even with a small amount of memory and even with a small data page cache, although it may perform poorly. Increasing the amount of memory available to Cloudscape and increasing the size of the data page cache improve performance.

In addition, you may want to prime *all* the caches in the background to make queries run faster when the user gets around to running them.

These caches include:

- the page (user data) cache (described above)

  Prime this cache by selecting from much-used tables that are expected to fit into the data page cache.
- the data dictionary cache

  The cache that holds information stored in the system tables. You can prime this cache with a query that selects from commonly used user tables.
- the statement cache

  The cache that holds connection-specific *Statements* (including *PreparedStatements*). You can prime this cache by preparing common queries ahead of time in a separate thread.
- the stored prepared statement cache

  The cache that holds system-wide stored prepared statements.

Tip Six. Tune the Size of Database Pages

Stick with 4K as the page size (the default, and the size operating systems use) unless:

- You are storing large objects.
- You have very large tables (over 10,000 rows).

  For very large tables, large pages reduces the number of I/Os required.
- You have very small tables and have footprint issues.

  For very small tables that will never grow, use 2K.
- Your application is read-only.

  For read-only applications, use a large page size (for example, 16K) with a *pageReservedSpace* of 0.

You may need to experiment with page size to find out what works best for your application and database.

Performance Trade-Offs of Large Pages

Using large database pages benefits database performance, notably decreasing I/O time. By default, the database page size is 4096 bytes. You can change the default database page size with the *cloudscape.storage.pageSize* property. For example:

```
cloudscape.storage.pageSize=8192
```

NOTE: Large database pages require more memory.

If row size is large, generally page size should be correspondingly large. If row size is small, page size should be small. Another rough guideline is to try to have at least 10 average-sized rows per page (up to 128K). Page size should not be less than 4K unless the table is very small and the whole base table can fit into one page.

Use a larger page size for tables with large columns or rows. After page size reaches 32K, each row inflicts a sightly higher overhead, so if the rows are small (say, < 100 bytes), it probably doesn't make sense to set the page size larger than 32K.

However, some applications involve rows whose size will vary considerably from user to user. In that situation, it is hard to predict what effect page size will have on performance.

If a table contains one large column along with several small columns, put the large column at the end of the row, so that commonly used columns won't be moved to overflow pages. Don't index large columns.

Large page size for indexes improves performance considerably.

When Large Page Size Does Not Improve Performance

- *Selective Queries*
  If your application's queries are very selective and use an index, large page size doesn't buy you much and potentially degrades performance because a larger page takes longer to read.

When Large Page Size Is Not Desirable

- *Limited memory*
  Large database pages reduce I/O time because Cloudscape can access more data with fewer I/Os. However, large pages require more memory.

Cloudscape allocates a bulk number of database pages in its page cache by default. If the page size is large, the system may run out of memory.

Here's a rough guideline: If the system is running Windows 95 and has more than 32 MB (or Windows NT and has more than 64 MB), it is probably beneficial to use 8K rather than 4K as the default page size.

Use the *-mx* flag as an optional parameter to the JVM to give the JVM more memory upon startup.

For example:

```
java -mx64 myApp
```

- *Limited disk space*
  If you can't afford the overhead of the minimum two pages per table, keep your page sizes small.
- *Large number of users*
  Very large page size reduces concurrency slightly when the system uses row-level locking.

Tip Seven. Avoid Expensive Queries

Some queries can, and should, be avoided. Two examples:

```
SELECT DISTINCT nonIndexedCol FROM HugeTable

SELECT * FROM HugeTable ORDER BY nonIndexedColumn
```

See "Prevent the User from Issuing Expensive Queries" on page 3-4.

Tip Eight. Use the Appropriate *getXXX* and *setXXX* Methods for the Type

JDBC is permissive. It lets you use *java.sql.ResultSet.getFloat* to retrieve an int, *java.sql.ResultSet.getObject* to retrieve any type, and so on. (*java.sql.ResultSet* and *java.sql.CallableStatemen*t provide *getXXX* methods and *java.sql.PreparedStatement* and *java.sql.CallableStatement* provide *setXXX* methods.) This permissiveness is convenient but expensive in terms of performance.

For performance reasons, use the recommended *getXXX* method when retrieving values, and use the recommended *setXXX* method when setting values for parameters.

Table 2-1 shows the recommended *getXXX* methods for given *java.sql* (JDBC) types, and their corresponding SQL-J types.

Table 2-1 Mapping of *java.sql.Types* to SQL-J Types

Recommended *getXXX* Method	*java.sql.Types*	SQL-J Types
getLong	BIGINT	LONGINT
getBytes	BINARY	BIT
getBoolean	BIT	BOOLEAN
getString	CHAR	CHAR
getDate	DATE	DATE
getBigDecimal	DECIMAL	DECIMAL
getDouble	DOUBLE	DOUBLE PRECISION
getDouble	FLOAT	DOUBLE PRECISION
getInt	INTEGER	INTEGER
getBinaryStream	LONGVARBINARY	LONG VARBINARY
getAsciiStream, getUnicodeStream	LONGVARCHAR	LONG VARCHAR
getBigDecimal	NUMERIC	DECIMAL
getObject	OTHER	Java classes
getFloat	REAL	REAL
getShort	SMALLINT	SMALLINT
getTime	TIME	TIME
getTimestamp	TIMESTAMP	TIMESTAMP
getByte	TINYINT	TINYINT
getBytes	VARBINARY	BIT VARYING
getString	VARCHAR	VARCHAR

Tip Nine. Tune Database Booting/Class Loading

By default, Cloudscape does not boot databases (and some core Cloudscape classes) in the system at Cloudscape startup but only at connection time. For multi-user systems, you may want to reduce connection time by booting one or all databases at startup instead. See "*cloudscape.system.bootAll*" on page 5-72 and "*cloudscape.database.noAutoBoot*" on page 5-24.

For embedded systems, you may want to boot the database in a separate thread (either as part of the startup, or in a connection request).

For more information, see “Shielding Users from Cloudscape Class-Loading Events” on page 3-10.

Tip Ten. Recompile Stale Stored Prepared Statements

Changes in the layout of data might require a new statement execution plan for old stored prepared statements. Some changes automatically force recompilation—adding or deleting indexes, for example. Other changes do not force recompilation—changes in the amount of data, for example. When a statement plan is no longer appropriate, we call it *stale*.

An egregious example of when changes in the amount of data make a statement plan go stale: You create a stored prepared statement for a SELECT statement against an empty table that has an index and try to use the same statement when the table has 20,000 rows. When the statement was created, Cloudscape would not have chosen an index as the access path in the statement execution plan. Recompiling the statement allows Cloudscape to choose a better statement execution plan.

Cloudscape automatically checks to see if a statement has gone stale after a given number of executions within a single Cloudscape session (100 by default). Systems in which data changes happen rapidly, or that shut down and restart often, may not benefit from this feature unless you configure the automatic checking to happen more often. For more details, see “Stale Plans and Stored Prepared Statements” on page 3-6.

NEW: Automatic recompilation of stale statements is new in Version 3.0.

Use RunTimeStatistics to see if Cloudscape is using a good statement execution plan. (See “Working with RunTimeStatistics” on page 3-18 for more information.)

Alternately, you can recompile stored prepared statements with the ALTER STATEMENT statement. UPDATES and DELETES may have similar issues.

No Longer in the Top Ten, but Worth Listing

Shut Down the System Properly

Cloudscape features crash recovery that restores the state of committed transactions in the event that the database exits unexpectedly, for example during a power failure. The recovery processing happens the next time the database is started after the unexpected exit. Your application can reduce the amount of work that the database has to do to start up the next time by shutting it down in an orderly fashion. See “Shutting Down Cloudscape or an Individual Database” on page 2-22 in the *Cloudscape Developer’s Guide*.

The Cloudscape utilities all perform an “orderly” shutdown.

Shorten Your Class Path

The structure of your class path can affect Cloudscape startup time and the time required to load a particular class.

The class path is searched linearly, so locate Cloudscape’s libraries at the beginning so that they are found first. If the class path points to a directory that contains multiple files, searching can be slow, because each file must be examined.

It is faster to search a zip or jar file than a directory.

Increase the Statement Cache Size

Cloudscape has a per-connection statement cache that it uses to try to avoid recompiling statements, prepared or not. If the text of an SQL request matches an already compiled statement in the cache, Cloudscape does not need to recompile the statement. If your application compiles *exactly the same query* more than once, you may want to increase the size of the statement cache to avoid this recompilation. By default, the statement cache size holds 20 statements. You can change this limit with the *cloudscape.language.statementCacheSize* property. For example:

```
cloudscape.language.statementCacheSize=100
```

NOTE: For situations in which you compile a prepared statement once and execute it many times, this property does not help performance. It helps only in situations in which you *compile* exactly the same statement more than once (either executing the same *Statement* more than once or preparing the same *PreparedStatement* more than once). If an application's statements are known in advance, it is better programming practice to compile those statements only once. This property is useful for those applications for which the statements are not known in advance. If the user happens to generate the same statement more than once in a session, the application can take advantage of the statement cache.

When Working with Development Tools, Pre-Load DatabaseMetaData Stored Prepared Statements

Some GUI Java development tools make heavy use of *DatabaseMetaData* method calls. Cloudscape can provide stored prepared statements that make these calls run much faster. By default, Cloudscape provides these stored prepared statements dynamically on an as-needed bases. When working with these tools, you should probably just pre-load the database with these stored prepared statements. You do this when creating the database. For information, see "*cloudscape.jdbc.metadataStoredPreparedStatements*" on page 5-29 in *Tuning Cloudscape*.

3 Tuning Databases and Applications

Chapter 2, “Performance Tips and Tricks”, provided some quick tips for improving performance. This chapter, while covering some of the same ground, provides more background on the basic design issues for improving performance. It also explains how to work with RunTimeStatistics and stored prepared statements.

- “Application and Database Design Issues” on page 3-1
- “Analyzing Statement Execution” on page 3-12
- “Storing Prepared Statements to Improve Performance” on page 3-13
- “Working with RunTimeStatistics” on page 3-18

Application and Database Design Issues

Things that you can do to improve the performance of Cloudscape applications fall into three categories. In order of importance, these categories are:

1. Avoiding Table Scans of Large Tables
2. Avoiding Compiling SQL Statements
3. Shielding Users from Cloudscape Class-Loading Events

Avoiding Table Scans of Large Tables

Cloudscape is fast and efficient, but when tables are huge, scanning tables may take longer than a user would expect. It’s even worse if you then ask Cloudscape to sort this data.

Things that you can do to avoid table scans fall into two categories. These categories are, in order of importance:

1. Index, Index, Index
2. Prevent the User from Issuing Expensive Queries
3. Understand When Statements Go Stale

Index, Index, Index

Have you ever thought what it would be like to look up a phone number in the phone book of a major metropolitan city if the book were not indexed by name? For example, to look up the phone number for John Jones, you couldn't go straight to the *J* page. You'd have to read the entire book. That's what a table scan is like. Cloudscape has to read the entire table to retrieve what you're looking for unless you create useful indexes on your table.

Create Useful Indexes

Indexes are useful when a query contains a WHERE clause. Without a WHERE clause, Cloudscape is *supposed* to return all the data in the table, and so a table scan is the correct (if not desirable) behavior. (More about that in "Prevent the User from Issuing Expensive Queries" on page 3-4.)

Cloudscape creates indexes on tables in the following situations:

- When you define a primary key, unique, or foreign key constraint on a table. See "CONSTRAINT clause" on page 1-29 of the *Cloudscape Reference Manual* for more information.
- When you explicitly create an index on a table with a CREATE INDEX statement.

For an index to be useful for a particular statement, one of the columns in the statement's WHERE clause must be the first column in the index's key.

NOTE: For a complete discussion of how indexes work and when they are useful (including pictures), see "What Is an Index?" on page 4-2 and "Index Use and Access Paths" on page 4-2.

Indexes provide some other benefits as well:

- If all the data requested are in the index, Cloudscape doesn't have to go to the table at all. (See "Covering Indexes" on page 4-7.)
- For operations that require a sort (ORDER BY), if Cloudscape uses the index to retrieve the data, it doesn't have to perform a separate sorting step

for some of these operations in some situations. (See “About the Optimizer’s Choice of Sort Avoidance” on page 4-18.)

NOTE: Don’t index on large columns.

Make Sure They Are Being Used, and Rebuild Them

If an index is useful for a query, Cloudscape is probably using it. However, you need to make sure. Analyze the way Cloudscape is executing your application’s queries. See “Analyzing Statement Execution” on page 3-12 for information on how to do this.

In addition, over time, index pages fragment. Rebuilding indexes improves performance significantly in these situations. To rebuild an index, drop it and then re-create it.

Think About Join Order

For some queries, join order can make the difference between a table scan (expensive) and an index scan (cheap). Here’s an example:

```
SELECT hotel_name
FROM Hotels, HotelAvailability
WHERE Hotels.hotel_id = HotelAvailability.hotel_id
AND Hotels.city_id = 10
```

If Cloudscape chooses *Hotels* as the outer table, it can use the index on *Hotels* to retrieve qualifying rows. (Given the data in *toursDB*, it will return three rows; three hotels have a *city_id* of 10.) Then it need only look up data in *HotelAvailability* three times, once for each qualifying row. And to retrieve the appropriate rows from *HotelAvailability*, it can use an index for *HotelAvailability*’s *hotel_id* column instead of scanning the entire table.

If Cloudscape chooses the other order, with *HotelAvailability* as the outer table, it will have to probe the *Hotels* table for *every row*, not just three rows, because there are no qualifications on the *HotelAvailability* table.

For more information about join order, see “Joins and Performance” on page 4-11.

Cloudscape usually chooses a good join order. However, as with index use, you should make sure. Analyze the way Cloudscape is executing your application’s queries. See “Analyzing Statement Execution” on page 3-12 for information on how to do this.

Prevent the User from Issuing Expensive Queries

Some applications have complete control over the queries that they issue; the queries are built into the applications. Other applications allow users to construct queries by filling in fields on a form. Any time you let users construct ad-hoc queries, you risk the possibility that the query a user constructs will be one like the following:

```
SELECT * FROM ExtremelyHugeTable
ORDER BY unIndexedColumn
```

This statement has no WHERE clause. It will require a full table scan. To make matters worse, Cloudscape will then have to order the data. It's likely that the user doesn't really want to browse through all 100,000 rows, and doesn't really care whether they're all in order.

Do everything you can to avoid table scans and sorting of large results (such as table scans).

Some things you can do to disallow such runaway queries:

- Use client-side checking to make sure some minimal fields are always filled in. Eliminate or disallow queries that cannot use indexes and are not optimizable. In other words, force an optimizable WHERE clause by making sure that the columns on which an index is built are included in the query's WHERE clause. Reduce or disallow DISTINCT clauses (which often require sorting) on large tables.
- For queries with large results, don't let the database do the ordering. Retrieve data in chunks (provide a Next button to allow the user to retrieve the next chunk, if desired), and order the data in the application.
- Don't do SELECT DISTINCT to populate lists; maintain a separate table of the unique items instead.

Understand When Statements Go Stale

- "Overview" on page 3-5
- "Automatic Stale Plan Invalidation" on page 3-5
- "Stale Plans and Stored Prepared Statements" on page 3-6
- "When a Change in Table Makes a Plan Stale" on page 3-7

Overview

When Cloudscape compiles and optimizes a statement, it creates what is called a statement execution plan. As discussed in “Avoiding Compiling SQL Statements” on page 3-8, this is a time-consuming process that you typically want to avoid. With stored prepared statements or prepared statements, Cloudscape can execute the same statement multiple times using the same plan and thus very quickly.

However, sometimes a statement’s existing plan may no be longer the best plan for executing the statement. These situations are caused by significant changes in the number of rows in a table. In these situations, you should be willing to pay the price of recompilation, because it is cheaper than using the stale execution plan.

For example, if you have an empty table or a table with only a few rows in it, Cloudscape’s optimizer will probably decide that it is easier to do a table scan than to access the data through an index. After you have loaded several thousand rows, however, a different statement execution plan is in order.

A statement can only go stale if it remains open or in the statement cache.

Table 3-1, “When Statements Can Go Stale,” on page 3-5, shows the situations in which it is technically possible for a statement to go stale.

Table 3-1 When Statements Can Go Stale

Type of Statement	When Statements Can Go Stale	Details
An application’s JDBC *Statement* or *PreparedStatement*	A statement can go stale when the Statement remains open or is retrieved out of the per-connection Statement cache.	A *Statement* (or *Prepared Statement*) can only exist within the context of a single Connection. Once the *Connection* is closed, all statements disappear. The next time the application connects, it must create the statement anew, and so Cloudscape creates a new statement plan.
A stored prepared statement (which is compiled when you create it)	A stored prepared statement can go stale any time after you create it.	A stored prepared statement’s plan persists across Connections and Cloudscape sessions.

Automatic Stale Plan Invalidation

To help you avoid stale statements, Cloudscape automatically checks whether a Statement’s execution plan is still appropriate after a certain number of executions

of that statement (by default, 100 executions) within a single Cloudscape session. Cloudscape determines if a statement's plan is appropriate by comparing the number of rows in the table with the number of rows in the table when the statement was first executed. If there has been a significant change in the number of the rows in the table, Cloudscape assumes the statement plan is no longer appropriate, and it invalidates and thus recompiles the statement.

This situation typically only affects SELECTs, UPDATEs, INSERT SELECTS, and DELETEs with WHERE clauses (statements that benefit from use of an index).

For example, imagine that you create a JDBC *Statement* (not a stored prepared statement). Its text follows:

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

Suppose that when your application creates this statement, the table is empty, and so Cloudscape chooses to scan the table instead of using the index on the *orig_airport* column.

During the course of a single *Connection*, the table gains a lot of data. The same or another user could insert thousands of rows. Unless Cloudscape notices the significant change in the size of the table, it continues to use the original—and now stale—plan. Given the system defaults, the hundredth time the application executes that statement, Cloudscape checks to see if the size of the table has changed significantly since the Statement was created. If it has, it marks the statement invalid and recompiles it before executing.

You can configure how often Cloudscape checks the appropriateness of a statement's plan with the *cloudscape.language.stalePlanCheckInterval* property.

Stale Plans and Stored Prepared Statements

Cloudscape tracks a table's size only within a single Cloudscape session. This limitation is not a problem for standard JDBC *Statements* and *PreparedStatements*, since they exist only within the scope of a single *Connection*. However, this may be a problem for stored prepared statements. If you stop and re-start Cloudscape often, Cloudscape may not notice a gradual change to a table that affects a stored prepared statement. In that situation, you may want to force recompilation, or you may want to configure Cloudscape to check whether to invalidate statements much more often (by setting *cloudscape.language.stalePlanCheckInterval* to a low value*)*. If you do not stop and re-start Cloudscape often (for example, in a server environment), Cloudscape's automatic invalidation and recompilation of stale plans works quite well even with the default value.

NEW: Automatic recompilation of stale statements is new in Version 3.0.

When a Change in Table Makes a Plan *Stale*

Cloudscape uses the following rules to determine whether a change in a table's size is significant enough to force recompilation for a statement:

For large tables, Cloudscape uses a simple percentage test. For smaller tables, Cloudscape uses a more complex, non-linear test. The smaller the table, the more sensitive Cloudscape is to changes.

- Simple percentage test (tables with 400 rows or more)

  If the number of rows in the table changes by more than ten percent, Cloudscape recompiles the statement.

  For example, if a table originally had 1000 rows when the statement was last compiled, and Cloudscape discovers that the table now has 1200 rows, it recompiles the statement, because the change in size is greater than ten percent. Similarly, if the table originally had 1000 rows, and it now has 800 rows, Cloudscape would recompile the statement. But if the table originally had 1000 rows and now has 1050 rows, the system would not recompile the statement.

- Complex percentage test (tables with fewer than 400 rows)

  Cloudscape takes the square of the difference in row counts and compares it to four times the original number of rows. If the square of the difference is greater than four times the original number of rows, it recompiles the statement.

NOTE: These rules may change in future releases.

Consider the following example:

> If a table originally had 8 rows, and the system discovers that it now has 15 rows, it recompiles the statement, because the square of the difference (49) is greater than four times the original size of the table (32). Similarly, if the table originally had 8 rows, and it now has 2 rows, it recompiles the statement, because the square of the difference (36) is greater than four times the original size of the table (32). On the other hand, if the table originally had 8 rows and now has 10 rows, it would not recompile the statement, because the square of the difference (4) is not greater than four times the original size of the table (32).

Avoiding Compiling SQL Statements

When you submit an SQL-J statement to Cloudscape, Cloudscape compiles and then executes the statement. *Compilation* is a time-consuming process that involves several steps, including optimization, the stage in which Cloudscape makes its statement execution plan. A statement execution plan includes whether to use an index, the join order, and so on.

Unless there are significant changes in the amount of data in a table or new or deleted indexes, Cloudscape will probably come up with the same statement execution plan for the same statement if you submit it more than once. This means that the same statements should share the same plan, and Cloudscape should not bother to recompile them. Cloudscape allows you to ensure this in the following ways (in order of importance):

1. Across Connections and Sessions
2. Within a Single Connection

Across Connections and Sessions

Statements from any connection or session can share the same statement execution plan (and avoid compilation) by using stored prepared statements. Stored prepared statements are database objects that persist within *and* across sessions. For more information, see “Storing Prepared Statements to Improve Performance” on page 3-13.

Stored prepared statements are not part of a standard but are unique to Cloudscape. Stored prepared statements are one of the most important features you can use to improve the performance of your application.

Within a Single Connection

As explained above in “Across Connections and Sessions” on page 3-8, you can create statements that can share the same statement execution plan (and avoid compilation) by using stored prepared statements. *Within a single connection*, you can also do the following to avoid extra compilation when for some reason it is impractical to use stored prepared statements:

- Your application can use *PreparedStatements* instead of *Statements.*

  PreparedStatements are JDBC objects that you prepare (compile) once and execute multiple times. If your application executes statements that are almost but not exactly alike, use *PreparedStatements*, which can contain

dynamic or IN parameters. Instead of using the literals for changing parameters, use ?s (placeholders) for these two parameters. Provide the values when you execute the statement.

PreparedStatements, unlike stored prepared statements, are a standard part of the JDBC API. Another way in which they are different from stored prepared statements is that they do not require writing to disk and do not create dictionary objects. For more information about using prepared statements, see “JDBC and the java.sql API” on page -7 (the listings for further reading on JDBC) in *Using the Cloudscape Documentation*.

For examples, see *JBMSTours.inserters.InsertCountries* in the *JBMSTours* sample application and the chapter on programming for performance in *Learning Cloudscape: The Tutorial*.

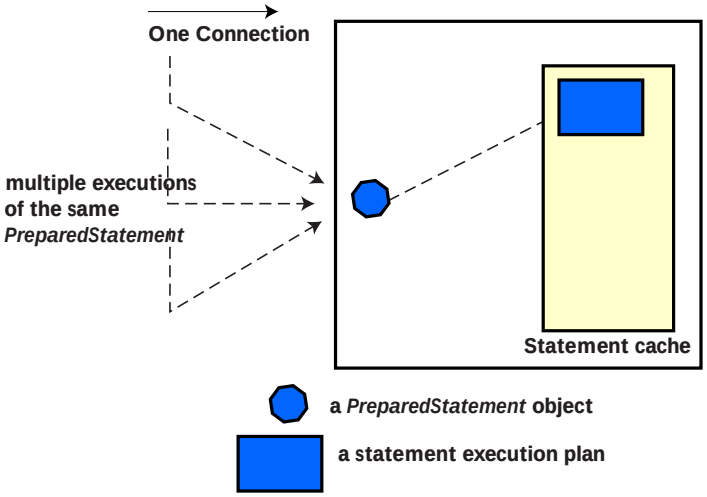


Figure 3-1 A connection need only compile a *PreparedStatement* once. Subsequent executions can use the same statement execution plan even if the parameter values are different. (*PreparedStatements* are not shared across connections.)

- Even if your statement uses *Statements* instead of *PreparedStatements*, Cloudscape can reuse the statement execution plan for the statements from the statement cache.

  When, in the same connection, an application submits an SQL *Statement* that has exactly the same text as one already in the cache, Cloudscape grabs the statement from the cache, even if the *Statement* has already been closed from the application. This is useful if your application allows the user to

create ad-hoc queries; if the user happens to run the same query more than once, Cloudscape can grab the statement out of the statement cache.

You can adjust the size of the statement cache. (See "*cloudscape.language.statementCacheSize*" on page 5-41.)

NOTE: The scroll type of the statement (forward-only vs. scrolling insensitive) of a statement must be identical, in addition to the SQL text, for there to be a match.

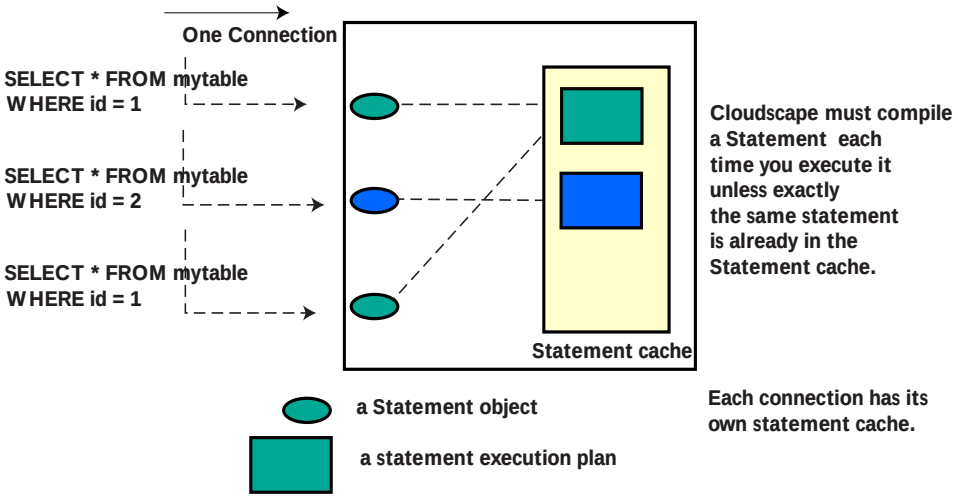


Figure 3-2 A connection can reuse a *Statement* object when the SQL text matches a prior statement *exactly*. (*PreparedStatements* are much more efficient.)

Shielding Users from Cloudscape Class-Loading Events

JVMs tend to load classes as they are needed, which means the first time you need a class in a piece of software, it takes longer to use.

Cloudscape has three clear cases when a lot of class loading occurs:

- *when the system boots*
  The system boots when you load the embedded driver, *COM.cloudscape.core.JDBCDriver*. In a server framework, the system boots when you start the server framework. Booting Cloudscape loads basic Cloudscape classes.

- *when the first database boots*
  Booting the first database loads some more Cloudscape classes. The default behavior is that the first database boots when the first connection is made to it. You can also configure the system to boot databases at startup. Depending on your application, one or the other may be preferable.
- *when you compile the first query*
  Compiling the first query loads additional classes. You can configure your system so that these classes are loaded when the database boots; when so configured, Cloudscape executes a built-in query synchronously at startup. See "*cloudscape.language.preloadClasses*" on page 5-36.

For any of these events, you can control the impact they have on users by starting them in separate threads while other tasks are occurring.

In addition, if you are using *PreparedStatements* instead of stored prepared statements, prepare them in a separate thread in the background while other tasks are occurring. Even if you are using stored prepared statements, you must still create a *Statement* or *PreparedStatement* that executes the stored prepared statement. Do that in a background thread as well.

Tuning Tips for Multi-User Systems

- For concurrency, use row-level locking and the READ_COMMITTED isolation level.
- For read-only applications, use table-level locking and the READ_COMMITTED isolation level.
- Boot databases at startup to minimize the impact of connecting.
- Where possible, create stored prepared statements. Query plans for stored prepared statements can be shared across connections.

Tuning Tips for Single-User Systems

- Cloudscape boots when you first load the embedded JDBC driver (*COM.cloudscape.core.JDBCDriver*). Load this driver during the least time-sensitive portion of your program, such as when it is booting or when you are waiting for user input. For server frameworks, the driver is loaded automatically when the server boots.
- Boot the database at connection (the default behavior), not at startup. Connect in a background thread if possible.
- Turn off row-level locking and use READ_COMMITTED isolation level. For single-connection applications, use SERIALIZABLE isolation level.

Analyzing Statement Execution

Once you create indexes, make sure that Cloudscape is using them. In addition, you may also want to find out the join order Cloudscape is choosing and force a better order if necessary.

Here is a general plan of attack for analyzing your application's SQL statements:

1. Collect your application's most frequently used SQL statements and transactions into a single test.
2. Create a benchmark test suite against which to run the sample queries. The first thing the test suite should do is checkpoint data (force Cloudscape to flush data to disk). You can do that with the following SQL-J statement:

```
CALL Factory.getDatabaseOfConnection().checkpoint()
```

NEW: The ability to checkpoint a database explicitly is new in Version 3.0.

3. Use performance timings to identify poorly performing queries. Try to distinguish between cached and uncached data. Focus on measuring operations on uncached data (data not already in memory). For example, the first time you run a query, Cloudscape returns uncached data. If you run the same query immediately afterward, Cloudscape is probably returning cached data. The performance of these two otherwise identical statements varies significantly and skews results.

4. Use RunTimeStatistics to identify tables that are scanned excessively. Are the appropriate indexes being used to satisfy the query? In some rare cases, you may need to force the index on the table. Is Cloudscape choosing the best join order? Force the best order, if appropriate. See "Working with RunTimeStatistics" on page 3-18 for instructions.
5. Make a change, then retest. (It's an iterative process.)
6. If changing data access does not create significant improvements, consider other database design changes, such as denormalizing data to reduce the number of joins required. Then review the tips in "Application and Database Design Issues" on page 3-1.

Storing Prepared Statements to Improve Performance

You can name and store prepared statements in a database. Storing prepared statements, which are already compiled, helps you avoid costly compilation time and speeds up query execution for the end user.

- "Creating a Stored Prepared Statement" on page 3-13
- "Executing a Stored Prepared Statement" on page 3-14
- "Additional Benefits for Multiple Connections: The Stored Prepared Statement Cache" on page 3-15
- "Invalid Stored Prepared Statements and Recompilation" on page 3-17
- "Stale Stored Prepared Statements" on page 3-18

Creating a Stored Prepared Statement

When you create a stored prepared statement with the CREATE STATEMENT command, Cloudscape prepares the statement, tags it with the name you give it, and stores the compiled code in a system table. For example:

```
CREATE STATEMENT getFullFlightInfo
AS SELECT *
FROM Flights
WHERE flight_id = ? AND segment_number = ?
```

NOTE: Create stored prepared statements after data are loaded into the referenced table, or recompile them after data is loaded. Creating a stored prepared statement when the referenced tables are empty leads to a bad plan. (See “Stale Stored Prepared Statements” on page 3-18.)

You may optionally provide sample values for parameters in the CREATE STATEMENT statement. This allows you to provide the optimizer with representative data so that the optimizer can choose a plan that will be suitable for the real values that will be used by the statement. Without these values, the optimizer makes guesses as to the selectivity of the parameters.

Providing sample values (with the USING clause) is particularly useful if you have a good idea of how the statement will be used in the future. For example:

```
CREATE STATEMENT getFullFlightInfo
AS SELECT *
FROM Flights
WHERE flight_id = ? AND segment_number = ?
USING VALUES (’AA1151’, 1)
```

Providing sample values with the USING clause is not useful for INSERTs (unless the INSERT also includes a SELECT). It is useful only for SELECT, UPDATE, or DELETE statements with WHERE clauses. This is because an INSERT statement will not get a different plan given differing amounts of data in the target table of the insert.

Executing a Stored Prepared Statement

After you create the stored prepared statement, you can invoke it using its name. Cloudscape can execute the statement without having to recompile it, unless you explicitly ask it to or some dependencies have been broken (see “Invalid Stored Prepared Statements and Recompilation” on page 3-17).

In your application, if the statement takes parameters, “prepare” an EXECUTE STATEMENT statement that references the stored prepared statement, then provide runtime values with the JDBC methods. Or, for development purposes, you can execute the statement with an EXECUTE STATEMENT statement with a USING clause, which allows you to provide the parameters directly.

If the statement does not take parameters, you can simply execute it. The “compilation cost” of an EXECUTE STATEMENT statement is the time it takes to retrieve the plan from the data dictionary and is unmeasurable after the first access.

For example:

```
// Execute a prepared statement via JDBC
PreparedStatement getFullFlightInfo = conn.prepareStatement(
    "EXECUTE GETFULLFLIGHTINFO");
// getFullFlightInfo now points to the stored statement statement
// Set the parameters using the setXXX() methods
getDirectFlights.setString(1, 'AA1111');
getDirectFlights.setInt(2, 1);
ResultSet rs = getFullFlightInfo.executeQuery();
while(rs.next()) {
/* retrieve data, etc. */ }
// try another one
getFullFlightInfo.setString(2, 'AA2222');
getDirectFlights.setInt(2, 1);
ResultSet rs = getFullFlightInfo.executeQuery();
```

Cloudscape allows you to provide parameters in the EXECUTE STATEMENT statement itself with another SQL statement such as a VALUES clause. This is useful when you are developing, but takes away many of the performance advantages of using stored prepared statements because of the cost of compiling the VALUES clause. Provide parameters with JDBC instead.

For more information about stored prepared statements and examples showing how to create and use them, see "CREATE STATEMENT statement" on page 1-45 and "EXECUTE STATEMENT statement" on page 1-73 of the *Cloudscape Reference Manual*.

Additional Benefits for Multiple Connections: The Stored Prepared Statement Cache

Stored prepared statements run faster than other kinds of statements because they allow you to avoid compilation. They also provide additional performance benefits in a multi-user situation because they reduce memory use. In a database with more than one *Connection*, each *Connection* shares the same object in memory for a stored prepared statement, caching only individual parameter information, not the entire object. This can improve performance significantly in multi-user databases because of the reduction in memory use.

There is one stored prepared statement cache per Cloudscape database. You can configure the size of the stored prepared statement cache; see "*cloudscape.language.spsCacheSize*" on page 5-37. For databases that have a lot of frequently used stored prepared statements, increase the size of this cache; the

default size is 32. The size of this cache never needs to be larger than the number of different stored prepared statements in a database. You can find out the number of stored prepared statements in a database with this query:

```
SELECT COUNT(*) FROM SYS.SYSSTATEMENTS
```

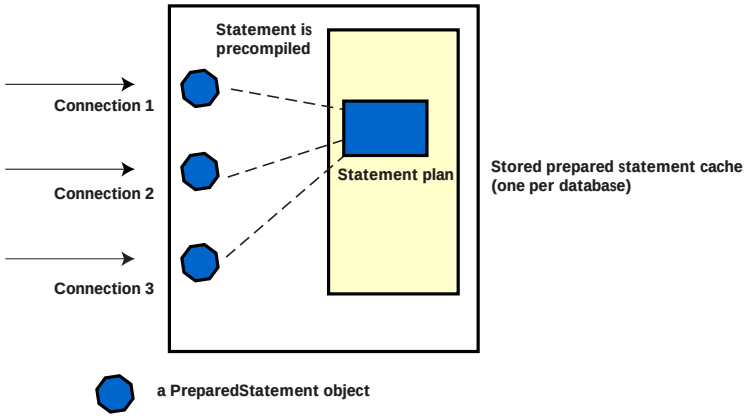


Figure 3-3 Multiple connections to a database can share the same statement execution plan and the same memory space when using stored prepared statements.

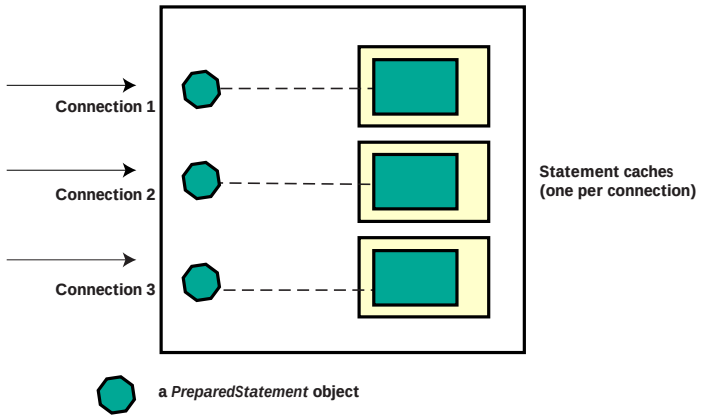


Figure 3-4 For *PreparedStatements*, each connection must compile the statement, and each connection must have a copy of the statement execution plan.

For examples of creating using stored prepared statements, see the JBMSTours sample application and the chapter on programming for performance in *Learning Cloudscape: The Tutorial*.

Invalid Stored Prepared Statements and Recompilation

Stored prepared statements, like other prepared statements, can become invalid. When a stored prepared statement is invalid, it is compiled the next time it is prepared or executed. Compilation happens automatically in the background and may be the cause of a slow execution. Once a stored prepared statement is re-compiled, however, it shouldn't need any more automatic recompilation unless it becomes invalid again.

A statement becomes *invalid* when one of its dependencies is broken through some kind of change in the database schema. For example, dropping an index, table, or view on which the statement depends breaks the statement's dependencies. As an example, let's look at the following stored prepared statement:

```
CREATE STATEMENT getFullFlightInfo
AS SELECT *
FROM Flights
WHERE flight_id = ? AND segment_number = ?
```

This statement depends on the following dictionary objects:

- the *Flights* table

  If you alter the table or add any new indexes, the statement becomes invalid.
- the index backing up the *Flights* table's primary key on *flight_id* and *segment_number*

  If you drop this index, the statement becomes invalid.

The *Cloudscape Reference Manual* details the dependencies for all dictionary objects and commands. Look for the heading "Dependency System" under the man page for a command. For an example, see the man page for "CREATE INDEX statement" on page 1-41 in the *Cloudscape Reference Manual.*

To avoid end-user inconvenience, make sure that no stored prepared statements are invalid before deploying your application and database. You must make sure that no stored prepared statements are invalid before deploying a read-only database. Invalid stored prepared statements require writing to disk, which will fail in a read-only environment.

You can automatically recompile all stored prepared statements with this command:

```
ALTER STATEMENT RECOMPILE ALL
```

You can automatically recompile all invalid stored prepared statements with this command:

```
ALTER STATEMENT RECOMPILE INVALID
```

You can recompile a single statement by specifying its name:

```
ALTER STATEMENT GetFullFlightInfo RECOMPILE
```

Stale Stored Prepared Statements

The preceding section, "Invalid Stored Prepared Statements and Recompilation" on page 3-17, discussed situations in which Cloudscape marked statements as invalid and automatically recompiled them. There are certain situations in which the statement is technically valid but its plan is *stale*, because it is no longer the best plan for executing the statement. These situations are caused not by changes in the database schema (which Cloudscape *does* detect), but by changes in the data layout.

For example, if you have an empty table or a table with only a few rows in it, Cloudscape's optimizer may decide that it is easier to do a table scan than to access the data through an index. After you have loaded several thousand rows, however, a different statement execution plan is in order.

Follow the directions in the preceding section, "Invalid Stored Prepared Statements and Recompilation" on page 3-17, to recompile all or individual stored prepared statements.

Working with RunTimeStatistics

Cloudscape provides a language-level tool for evaluating the performance and the execution plans of statements, the RUNTIMESTATISTICS() built-in function.

- "Overview" on page 3-19
- "How You Use It" on page 3-19
- "Analyzing the Information" on page 3-20

Overview

When a special attribute (RunTimeStatistics) is turned on for a connection, Cloudscape creates an object that implements the *COM.cloudscape.types.RunTimeStatistics* interface for each statement executed within the *Connection* until the attribute is turned off.

For the most recently executed query, the object displays information about:

- *the length of the compile time and the execute time*
  This can help in benchmarking queries.
- *the statement execution plan*
  This is a description of result set nodes, whether an index was used, what the join order was, how many rows qualified at each node, and how much time was spent in each node. This information can help you determine whether you need to add indexes or rewrite queries.

The exact details presented, as well as the format of presentation, may change.

You access the object created with the RUNTIMESTATISTICS() built-in function. You generally do not work with the object directly, but instead call one of its methods directly on the function. See "RUNTIMESTATISTICS()" on page 1-161 in the *Cloudscape Reference Manual*.

How You Use It

NOTE: Cloudview presents an easy-to-use interface for runtime statistics. See "Viewing Runtime Statistics in Cloudview" on page 3-24.

These are the basic steps for working with the RUNTIMESTATISTICS() function.

1. Turn on the RunTimeStatistics attribute with the SET RUNTIMESTATISTICS statement command (see "SET RUNTIMESTATISTICS statement" on page 1-111 in the *Cloudscape Reference Manual* for more information).

   ```
   -- turns on RunTimeStatistics
   SET RUNTIMESTATISTICS ON
   ```

2. Turn on the Statistics Timing attribute with the SET STATISTICS TIMING statement command (see "SET STATISTICS TIMING statement" on page 1-113 of the *Cloudscape Reference Manual* for more information). If you do not turn on this attribute, you will see the statement execution plan only, and not the timing information.

```
SET STATISTICS TIMING ON
```

3. If you are working in ij, set the display width to 5000 or another high number.
4. Execute an SQL-J statement.
5. Retrieve the values from one or more of the methods of the *RunTimeStatistics* interface executed against the RUNTIMESTATISTICS() built-in function. For example:

   ```
   VALUES RUNTIMESTATISTICS().toString()
   ```

   For a complete list of the methods available on the interface, see the javadoc for *COM.cloudscape.types.RunTimeStatistics*.

   In a multi-threaded environment, you must synchronize steps 4 and 5.
6. Turn off RunTimeStatistics and Statistics Timing.

These steps have shown you how you would work with RunTimeStatistics within ij. The basic steps for working with RunTimeStatistics are the same in a java program. For a complete coding example, see the sample program *JBMSTours.RunTime* in the JBMSTours application. Run the program for examples of several different kinds of statements and the different output they produce.

NOTE: The exact content and format of the statement execution plan are subject to change.

Analyzing the Information

Statistics Timing

If the Statistics timing attribute is on, the *RunTimeStatistics* object provides information about how long each stage of the statement took. An SQL-J statement has two basic stages within Cloudscape: compilation and execution. Compilation is the work done while the statement is prepared. Compilation is composed of the following stages: parsing, binding, optimization, and code generation. Execution is the actual evaluation of the statement. (If the statistics timing attribute is off, it shows a zero time for each stage.)

Statement Execution Plan

The *RunTimeStatistics* object also provides information about the *statement execution plan*. A statement execution plan is composed of a tree of result set nodes. A result set node represents the evaluation of one portion of the statement; it returns rows to a calling (or parent) node and can receive rows from a child node. A node can have one or more children. Starting from the top, if a node has children, it requests rows from the children. Usually only the execution plans of DML statements (queries, inserts, updates, and deletes, not dictionary object creation) are composed of more than one node.

For example, consider the following query:

```
SELECT *
FROM Countries
```

This simple query involves one node only—reading all the data out of the *Countries* table. It involves a single node with no children. This result set node is called a *TableScanResultSet*. RunTimeStatistics' text for this node looks something like this:

```
TableScanResultSet for COUNTRIES at read committed isolation
level using share row locking chosen by the optimizer.
numOpens = 1
rowsSeen = 115
rowsFiltered = 0
fetchSize = 16
        constructor time (milliseconds) = 0
        open time (milliseconds) = 10
        next time (milliseconds) = 30
        close time (milliseconds) = 20
        next time in milliseconds/row = 0
scanInfo:
        columnsFetchedBitSet=cols:(all)
        numColumnsFetched=3
        numPagesVisited=2
        numRowsQualified=115
        numRowsVisited=115
        scanType=heap
        startPosition:
null    stopPosition:
null    qualifiers:
None
        optimizer estimated row count:          124.00
        optimizer estimated cost:          50.38
```

Consider this second, more complex query:

```
SELECT Country
FROM Countries
WHERE Region = 'Central America'
```

When executed, this query involves two nodes—one to retrieve qualifying rows (the restriction is done at this node) and one to project the requested columns. So, at bottom, there is a *TableScanResultSet* for scanning the table. The qualifier (Region = 'Central America') is evaluated in this node. These data are passed up to the parent node, called a *ProjectRestrictResultSet*, in which the rows are projected—only the *country* column is needed (the first column in the table). RunTimeStatistics' text for these two nodes looks something like this:

```
ProjectRestrictResultSet (1):
numOpens = 1
rowsSeen = 6
rowsFiltered = 0
        constructor time (milliseconds) = 0
        open time (milliseconds) = 0
        next time (milliseconds) = 10
        close time (milliseconds) = 0
        restriction time (milliseconds) = 0
        projection time (milliseconds) = 0
Source result set:
        TableScanResultSet for COUNTRIES at read committed
isolation level using share row locking chosen by the
optimizer.
        numOpens = 1
        rowsSeen = 6
        rowsFiltered = 0
        fetchSize = 16
                constructor time (milliseconds) = 10
                open time (milliseconds) = 0
                next time (milliseconds) = 10
                close time (milliseconds) = 0
                next time in milliseconds/row = 1
        scanInfo:
                columnsFetchedBitSet=cols:{0, 2}
                numColumnsFetched=2
                numPagesVisited=2
                numRowsQualified=6
                numRowsVisited=115
                scanType=heap
```

```
                startPosition:
null            stopPosition:
null            qualifiers:
Column[0] Id: 2
Operator: =
Ordered nulls: false
Unknown return value: false
Negate comparison result: false
                optimizer estimated row count:
11.50
                optimizer estimated cost:          49.22
```

Other, more complex queries such as joins and unions have other types of result set nodes.

For inserts, updates, and deletes, rows flow out of the top, where they are inserted, updated, or deleted. For selects (queries), rows flow out of the top into a result set that is returned to the user.

Table 1-4 on page 1-164 in the *Cloudscape Reference Manual* shows the many possible *ResultSet* nodes that might appear in an execution plan. Some examples are:

- *ProjectRestrictResultSet*
- *SortResultSet*
- *TableScanResultSet*
- *IndexScanResultSet*
- *IndexRowToBaseRowResultSet*

In addition, read Chapter 4, “DML Statements and Performance”, for more information about some of the ways in which Cloudscape executes statements.

The statement execution plan shows how long each node took to evaluate, how many rows were retrieved, whether an index was used, and so on. If an index was used, it shows the start and stop positions for the matching index scan. Looking at the plan may help you determine whether to add an index or to rewrite the query.

Subset of Statement Execution Plan

You can execute a method called *getScanStatisticsText* on the *RunTimeStatistics* interface that returns a string describing only a subset of the full statement execution plan. It provides information only about those nodes that access a table or index. Using this method is the easiest way to find out whether Cloudscape used an index or scanned the entire table and what the join order was.

Optimizer Estimates

Runtime statistics show the optimizer estimates for a particular node. They show the optimizer's estimated row count and the optimizer's "estimated cost."

The *estimated row count* is the query optimizer's estimate of the number of qualifying rows for the table or index for the entire life of the query. If the table is the inner table of a join, the estimated row count will be for all the scans of the table, not just for a single scan of the table.

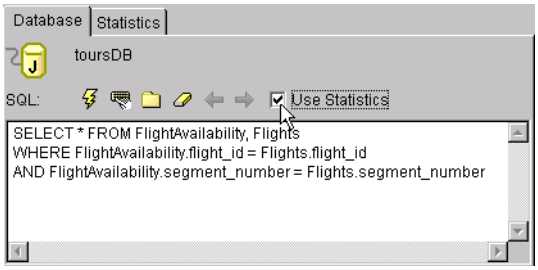
NEW: You can execute a method called *getEstimatedRowCount*, which returns the optimizer's estimated row count. (For a tip that explains when that might be useful, see "Use RunTimeStatistics to Get an Approximate Row Count for a Table" on page 10-15 in the *Cloudscape Developer's Guide*.)

The *estimated cost* consists of a number, which is a relative number; it does not correspond directly to any time estimate. It is not, for example, the number of milliseconds or rows. Instead, the optimizer constructs this number for each possible access path. It compares the numbers and chooses the access path with the smallest number.

Viewing Runtime Statistics in Cloudview

Cloudview makes working with runtime statistics easy. To use it:

1. Before executing the query you want to analyze, select Use Statistics (the equivalent of issuing the SET RUNTIMESTATISTICS ON and SET STATISTICS TIMING ON commands).



2. Execute the query.
3. Select the Statistics tab (equivalent to issuing VALUES RUNTIMESTATISTICS()).
4. An object inspection window appears. *ResultSet* nodes appear in a tree format.

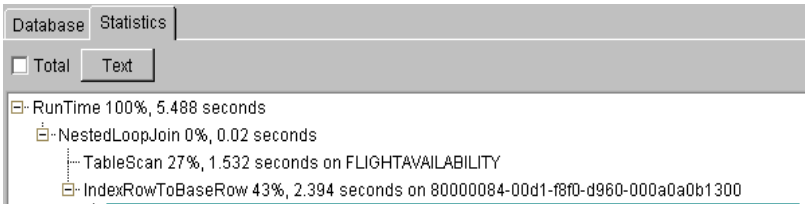


Figure 3-5 *ResultSet* nodes displayed by Cloudview

Figure 3-5 shows the *ResultSet* nodes for the query used as an example in "Join Order Case Study" on page 4-17. It allows you to see at a glance that the query was executed as a nested loop join; that *FlightAvailability* is the outer table in the join (because it is the first *ResultSet*); and that Cloudscape used an index to look up values on the inner table, *Flights* (because there is an Index scan on *Flights*, with the name of the index). *IndexRowToBaseRowResultSet* is the node in which Cloudscape uses the row ID it got from the index key in the index scan to return the matching row from the base table.

5 Cloudview displays timing statistics for each node. If you select Total, near the top of the screen, Cloudview displays the time spent in the node, including the time for all the node's children. For example, using the example in Figure 3-5 on page 3-25, the *IndexRowToBaseRowResultSet* is part of (a child of) the *NestedLoopJoin* node. Selecting *Total* adds the time taken to execute this child node, along with any other, to the timing for the *NestedLoopJoin* node.

4 DML Statements and Performance

- "Performance and Optimization" on page 4-1
- "Locking and Performance" on page 4-28

Performance and Optimization

A DBMS often has a choice about the access path for retrieving data. For example, the DBMS may use an index (fast lookup for specific entries) or scan the entire table to retrieve the appropriate rows. In addition, in statements in which two tables are joined, the DBMS can choose which table to examine first. *Optimization* means that DBMS makes the best (optimal) choice of access paths and join order. True query optimization means that the DBMS will usually make a good choice regardless of how the query is written. The optimizer does not necessarily make the *best* choice, just a good one.

Cloudscape's query optimizer can use indexes to improve the performance of DML (data manipulation language) statements such as queries, updates, and deletes. It also makes decisions about join order, type of join, and a few other matters.

This section gives an overview of the Cloudscape optimizer and discusses performance issues in the execution of DML statements.

This section covers the following topics:

- "Index Use and Access Paths" on page 4-2
- "Joins and Performance" on page 4-11
- "Cloudscape's Cost-Based Optimization" on page 4-14

Index Use and Access Paths

If you define an index on a column or columns, the query optimizer can use the index to find data in the column more quickly. Cloudscape automatically creates indexes to back up primary key, foreign key, and unique constraints, so those indexes are always available to the optimizer, as well as those that you explicitly create with the CREATE INDEX command. The way Cloudscape gets to the data—via an index or directly via the table—is called the *access path*.

This section covers the following topics:

- "What Is an Index?" on page 4-2
- "What's Optimizable?" on page 4-5
- "Covering Indexes" on page 4-7
- "Useful Indexes Can Use Qualifiers" on page 4-9
- "When a Table Scan Is Better" on page 4-10

What Is an Index?

An index is a database structure that provides quick lookup of data in a column or columns of a table. It is probably best described through examples.

For example, the *Flights* table in the *toursDB* database has three indexes:

- one on the *orig_airport* column (called *OrigIndex*)
- one on the *dest_airport* column (called *DestIndex*)
- one enforcing the *primary key* constraint on the *flight_id* and *segment_numbe*r columns (which has a system-generated name)

This means there are three separate structures that provide shortcuts into the *Flights* table. Let's look at one of those structures, *OrigIndex*. Table 4-1 shows a portion of the *Flights* table from *toursDB*.

	FLIGHT_ID	SE...	ORIG_AIRPORT	DEPART_TIME	DEST_AIRPORT	ARRIVE_TIME	MEAL	FLYIN...	MILES	AIRCRA...
0	AA1111	1	ABQ	09:00:00	LAX	09:19:00	L	1.328000	664	B747
1	AA1112	1	LAX	09:00:00	ABQ	11:19:00	L	1.328000	664	B747
2	AA1113	1	ABQ	09:00:00	PHX	09:39:00	L	0.658	329	B747
3	AA1114	1	PHX	09:00:00	ABQ	09:39:00	L	0.658	329	B747
4	AA1115	1	ABQ	09:00:00	OKC	11:02:00	L	1.034	517	B747
5	AA1116	1	OKC	09:00:00	ABQ	09:02:00	L	1.034	517	B747
6	AA1117	1	AKL	09:00:00	HNL	18:48:00	L	8.804	4402	B747
7	AA1118	1	HNL	13:30:00	AKL	21:18:00	L	8.804	4402	B747
8	AA1119	1	AKL	09:00:00	NRT	15:59:00	L	10.996000	5498	B747
9	AA1120	1	NRT	09:00:00	AKL	23:59:00	L	10.996000	5498	B747
10	AA1121	1	AKL	09:00:00	SYD	09:40:00	L	2.682000	1341	B747
11	AA1122	1	SYD	09:00:00	AKL	13:40:00	L	2.682000	1341	B747
12	AA1123	1	ALB	09:00:00	LGA	09:16:00	L	0.27	135	B747
13	AA1124	1	LGA	09:00:00	ALB	09:16:00	L	0.27	135	B747

Figure 4-1 Partial view of the *Flights* table

OrigIndex 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 4-2:

- For every row in *Flights*, there is an entry in *OrigIndex* 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 set of entries constitutes the leaf level of the index's BTREE structure.
- One or more abstract levels in the BTREE structure have values that point into a lower level of the index, much as tab dividers in a three-ring notebook help you find the correct section quickly. The most abstract level of the index's BTREE structure is called the root level. These levels help Cloudscape determine where to begin an index scan.

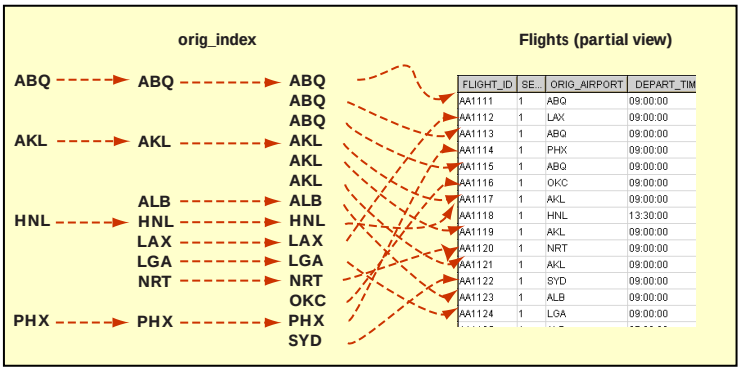


Figure 4-2 The index on the *orig_airport* column helps Cloudscape find the rows for which *orig_airport* is equal to a specific value.

When an index includes more than one column, the first column is the main one by which the entries are ordered. For example, the index on (*flight_id*, *segment_number*) is ordered first by *flight_id*. If there is more than one *flight_id* of

the same value, those entries are then ordered by *segment_number*. An excerpt from the entries in the index might look like this:

```
'AA1111' 1
'AA1111' 2
'AA1112' 1
'AA1113' 1
'AA1113' 2
```

Indexes are helpful only sometimes. This particular index is useful when a statement's WHERE clause is looking for rows for which the value of *orig_airport* is some specific value or range of values. SELECTs, UPDATEs, and DELETEs can all have WHERE clauses.

For example, *OrigIndex* is helpful for statements such as the following:

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

SELECT *
FROM Flights
WHERE orig_airport < 'BBB'

SELECT *
FROM Flights
WHERE orig_airport >= 'MMM'
```

DestIndex is helpful for statements such as the following:

```
SELECT *
FROM Flights
WHERE dest_airport = 'SCL'
```

The primary key index (on *flight_id* and *segment_number*) is helpful for statements such as the following:

```
SELECT *
FROM Flights
WHERE flight_id = 'AA1111'

SELECT *
FROM Flights
WHERE flight_id BETWEEN 'AA1111' AND 'AA1115'

SELECT *
FROM FlightAvailability AS fa, Flights AS fts
WHERE flight_date > CURRENT_DATE
```

```
AND fts.flight_id = fa.flight_id
AND fts.segment_number = fa.segment_number
```

The next section discusses why the indexes are helpful for these statements but not for others.

What's Optimizable?

As you learned in the previous section, Cloudscape may be able to use an index on a column to find data more quickly. If Cloudscape can use an index for a statement, that statement is said to be *optimizable*. The statements shown in the preceding section allow Cloudscape to use the index because their WHERE clauses provide start and stop conditions. That is, they tell Cloudscape the point at which to begin its scan of the index and where to end the scan.

For example, a statement with a WHERE clause looking for rows for which the *orig_airport* value is less than *BBB* means that Cloudscape must begin the scan at the beginning of the index; it can end the scan at *BBB*. This means that it avoids scanning the index for most of the entries.

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

NOTE: A WHERE clause can have more than one part. Parts are linked with the word *AND* or *OR*. Each part is called a *predicate*. WHERE clauses with predicates joined by OR are not optimizable. WHERE clauses with predicates joined by AND are optimizable if *at least one* of the predicates is optimizable. For example:

```
SELECT * FROM Flights
WHERE flight_id = 'AA1111' AND
segment_number <> 2
```

In this example, the first predicate is optimizable; the second predicate is not. Therefore, the statement is optimizable.

NOTE: In a few cases, a WHERE clause with predicates joined by OR can be transformed into an optimizable statement. See "OR Transformations" on page A-6.

Directly Optimizable Predicates

Some predicates provide clear-cut starting and stopping points. A predicate provides start or stop conditions, and is therefore optimizable, when:

- It uses a simple column reference to a column (the name of the column, not the name of the column within an expression or method call). For example, the following is a simple column reference:

  ```
  WHERE orig_airport = 'SFO'
  ```

  The following is not:

  ```
  WHERE orig_airport.toString() = 'SFO'
  ```

- It refers to a column that is the first or only column in the index.

  References to *contiguous* columns in other predicates in the statement when there is a multi-column index can further define the starting or stopping points. (If the columns are not contiguous with the first column, they are not optimizable predicates but can be used as *qualifiers*.) For example, given a composite index on *FlightAvailability* (*flight_id*, *segment_number,* and *flight_date*), the following predicate satisfies that condition:

  ```
  WHERE flight_id = 'AA1200' AND segment_number = 2
  ```

  The following one does not:

  ```
  WHERE flight_id = 'AA1200' AND flight_date = CURRENT_DATE
  ```

- The column is compared to a *constant* or to an expression that does not include columns in the same table. Examples of valid expressions: *other_table.column_a*, *?* (dynamic parameter), *7+9*. The comparison must use the following operators:

  - =
  - <
  - <=
  - >
  - >=
  - IS NULL

Indirectly Optimizable Predicates

Some predicates are transformed internally into ones that provide starting and stopping points and are therefore optimizable.

Predicates that use the following comparison operators can be transformed internally into optimizable predicates:

- BETWEEN

- LIKE (in certain situations)
- IN (in certain situations)

For details on these and other transformations, see Appendix A, “Internal Language Transformations”.

Joins

Joins specified by the JOIN keyword are optimizable. This means that Cloudscape can use an index on the inner table of the join (start and stop conditions are being supplied implicitly by the rows in the outer table).

Note that joins built using traditional predicates are also optimizable. For example, the following statement is optimizable:

```
SELECT * FROM Countries, Cities
WHERE Countries.country_ISO_code = Cities.country_ISO_code
```

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 *covers the query* if all the columns specified in the query are part of 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 at all, but can retrieve all data through index access alone. For example, in the following queries, *OrigIndex* covers the query:

```
SELECT orig_airport
FROM Flights

SELECT DISTINCT orig_airport.toLowerCase()
FROM Flights
```

Cloudscape can get all required data out of the index instead of from the table.

Single-Column Index Examples

The following queries do *not* provide start and stop conditions for a scan of *OrigIndex*, the index on the *orig_airport* column in *Flights*. However, some of these queries allow Cloudscape to do an index rather than a table scan because the index covers the query.

```
-- Cloudscape would scan entire table; comparison is not with a
-- constant or with a column in another table
```

```
SELECT *
FROM Flights
WHERE orig_airport = dest_airport

-- Cloudscape would scan entire table; <> operator is not
optimizable
SELECT *
FROM Flights
WHERE orig_airport <> 'SFO'

-- not valid operator for matching index scan
-- However, Cloudscape would do an index
-- rather than a table scan because
-- index covers query
SELECT orig_airport
FROM Flights
WHERE orig_airport <> 'SFO'

-- method invocation is not simple column reference
-- Cloudscape would scan entire index, but not table
-- (index covers query)
SELECT orig_airport
FROM Flights
WHERE orig_airport.toLowerCase() = 'sfo'
```

Multiple-Column Index Example

The following queries do provide start and stop conditions for a scan of the primary key index on the *flight_id* and *segment_number* columns in *Flights*:

```
-- the where clause compares both columns with valid
-- operators to constants
SELECT *
FROM Flights
WHERE flight_id = 'AA1115'
AND segment_number < 2

-- the first column is in a valid comparison
SELECT *
FROM Flights
WHERE flight_id < 'BB'

-- LIKE is transformed into >= and <=, providing
-- start and stop conditions
SELECT *
FROM Flights
WHERE flight_id LIKE 'AA%'
```

The following queries do not:

```
-- segment_number is in the index, but it's not the first column;
-- there's no logical starting and stopping place
SELECT *
FROM Flights
WHERE segment_number = 2

-- Cloudscape would scan entire table; comparison of first column
-- is not with a constant or column in another table
-- and no covering index applies
SELECT *
FROM Flights
WHERE orig_airport = dest_airport
AND segment_number < 2
```

Useful Indexes Can Use Qualifiers

Matching index scans can use qualifiers that further restrict the result set. Remember that a WHERE clause that contains at least one optimizable predicate is optimizable. Nonoptimizable predicates may be useful in other ways.

Consider the following query:

```
SELECT *
FROM FLIGHTS
WHERE orig_airport < 'BBB'
AND orig_airport <> 'AKL'
```

The second predicate is not optimizable, but the first predicate is. The second predicate becomes a qualification for which Cloudscape evaluates the entries in the index as it traverses it.

- The following comparisons are valid qualifiers:
  - =
  - <
  - <=
  - >
  - >=
  - IS NULL
  - BETWEEN
  - LIKE
  - <>

  - IS NOT NULL
- The qualifier's reference to the column does not have to be a simple column reference; you can put the column in an expression.
- The qualifier's column does not have to be the first column in the index and does not have to be contiguous with the first column in the index.

When a Table Scan Is Better

Sometimes a table scan is the most efficient way to access data, even if a potentially useful index is available. For example, if the statement returns virtually all the data in the table, it is more efficient to go straight to the table instead of looking values up in an index, because then Cloudscape is able to avoid the intermediate step of retrieving the rows from the index lookup values.

For example:

```
SELECT *
FROM Flights
WHERE dest_airport < 'Z'
```

In the *Flights* table, most of the airport codes begin with letters that are less than *Z*. Depending on the number of rows in the table, it is probably more efficient for Cloudscape to go straight to the table to retrieve the appropriate rows. However, for the following query, Cloudscape uses the index:

```
SELECT *
FROM Flights
WHERE dest_airport < 'B'
```

Only a few flights have airport codes that begin with a letter less than *B*.

Indexes Have a Cost for Inserts, Updates, and Deletes

Cloudscape has to do work to maintain indexes. If you insert into or delete from a table, the system has to insert or delete rows in all the indexes on the table. If you update a table, the system has to maintain those indexes that are on the columns being updated. So having a lot of indexes can speed up select statements, but slow down inserts, updates, and deletes.

NOTE: Updates and deletes with WHERE clauses can use indexes for scans, even if the indexed column is being updated.

Updatable cursors cannot use indexes on updatable columns.

How Indexes Affect Cursors

For updatable cursors, the query optimizer avoids any index that includes an updatable column. Specifying a list of column names in the FOR UPDATE clause allows the optimizer to choose an index on any column not specified. To restrict the columns that are updatable through the cursor, you can specify a list of column names in the FOR UPDATE clause.

For example, the following statement cannot use an index during execution and instead must scan the entire table:

```
-- Cloudscape cannot use the index on flight_id because
-- all columns are updatable
SELECT *
FROM Flights
WHERE flight_id = 'AA1113'
FOR UPDATE
```

However, the following statement can use indexes:

```
-- Cloudscape can use the index on flight_id because it is not an
-- updatable column
SELECT *
FROM Flights
WHERE flight_id = 'AA1113'
FOR UPDATE OF depart_time, arrive_time
```

Joins and Performance

Joins, SQL-J statements in which Cloudscape selects data from two or more tables using one or more key columns from each table, can vary widely in performance. Factors that affect the performance of joins are join order, indexes, and join strategy.

Join Order Overview

The Cloudscape optimizer usually makes a good choice about join order. This section discusses the performance implications of join order.

In a join operation involving two tables, Cloudscape scans the tables in a particular order. Cloudscape accesses rows in one table first, and this table is now called the *outer table*.

Then, for each qualifying row in the outer table, Cloudscape looks for matching rows in the second table, which is called the *inner table*.

Cloudscape accesses the outer table once, and the inner table probably many times (depending on how many rows in the outer table qualify).

This leads to a few general rules of thumb about join order:

- If the join has no restrictions in the WHERE clause that would limit the number of rows returned from one of the tables to just a few, the following rules apply:
  - If *only one* table has an index on the joined column or columns, it is much better for that table to be the inner table. This is because for each of the many inner table lookups, Cloudscape can use an index instead of scanning the entire table, as shown in Figure 4-3.

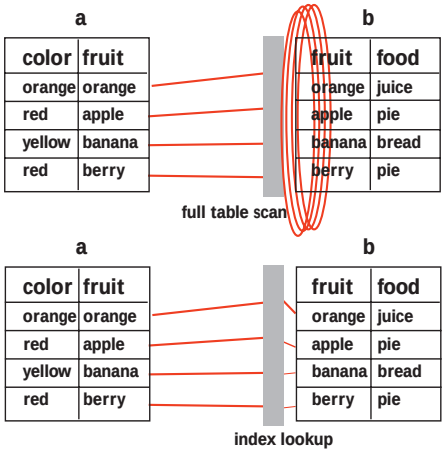


Figure 4-3 In a query that joins table *a* with table *b* with no restrictions in a WHERE clause, Cloudscape scans table *a* once, and for each qualifying row (four rows), it looks for data in table *b*. If there is no useful index on table *b*, Cloudscape scans the entire table each time. If there is a useful index on table *b* (on the joining column, fruit), Cloudscape can do a quick index lookup for each row (four lookups).

  - Since indexes on inner tables are accessed many times, if the index on one table is smaller than the index on another, the table with the smaller one should probably be the inner table. That is because smaller indexes (or tables) can be cached (kept in Cloudscape's memory, allowing Cloudscape to avoid expensive I/O for each iteration).
- On the other hand, if a query has restrictions in the WHERE clause for one table that would cause it to return only a few rows from that table (for example, WHERE flight_id = 'AA1111'), it is better for the restricted table

to be the outer table. Cloudscape will have to go to the inner table only a few times anyway.

Consider:

```
SELECT *
FROM huge_table, small_table
WHERE huge_table.unique_column = 1
AND huge_table.other_column = small_table.non_unique_column
```

In this case, the qualification *huge_table.unique_column = 1* (assuming a unique index on the column) qualifies only one row, so it is better for *huge_table* to be the outer table in the join.

Join Strategies

The default join strategy in Cloudscape is called a *nested loop*. For each qualifying row in the outer table, Cloudscape uses the appropriate access path (index or table) to find the matching rows in the inner table.

Another available type of join in Cloudscape is called a *hash* join. For joins of this type, Cloudscape constructs a hash table representing all the selected columns of the inner table. For each qualifying row in the outer table, Cloudscape does a quick lookup on the hash table to get the inner table data. Cloudscape has to scan the inner table or index only once, to build the hash table.

Nested loop joins are preferable in most situations.

Hash joins are preferable in situations in which the inner table values are unique and there are many qualifying rows in the outer table. Hash joins require that the statement's WHERE clause be an optimizable equijoin:

- It must use the = operator to compare column(s) in the outer table to column(s) in the inner table.
- References to columns in the inner table must be simple column references. Simple column references are described in "Directly Optimizable Predicates" on page 4-5.

The hash table for a hash join is held in memory and if it gets big enough, it can cause the JVM to run out of memory. The optimizer makes a very rough estimate of the amount of memory required to make the hash table. If it estimates that the amount of memory required would exceed 1 MB, the optimizer chooses a nested loop join instead.

NEW: Beginning in Version 3.0, Cloudscape allows multiple columns as hash keys. This is a significant performance improvement. Another improvement in Version 3.0 is that the inner table of a hash join no longer

needs to be a base table (instead of a view, derived table, etc.). This expands the number of situations in which Cloudscape can choose a hash join.

Cloudscape's Cost-Based Optimization

In Version 3.0, the query optimizer makes cost-based decisions to determine:

- Which index (if any) to use on each table in a query (see "About the Optimizer's Choice of Access Path" on page 4-14)
- The join order (see "About the Optimizer's Choice of Join Order" on page 4-16)
- The join strategy (new in 2.0) (see "About the Optimizer's Choice of Join Strategy" on page 4-18)
- Whether to avoid additional sorting (new in 2.0) (see "About the Optimizer's Choice of Sort Avoidance" on page 4-18)
- Automatic lock escalation (new in 2.0) (see "About the Optimizer's Selection of Lock Granularity" on page 4-21)
- Whether to use bulk fetch (see "About the Optimizer's Selection of Bulk Fetch" on page 4-24)

About the Optimizer's Choice of Access Path

In deciding whether to use an index or to do a table scan, the optimizer estimates the number of rows that will be read (see "When a Table Scan Is Better" on page 4-10). Table scans always require scanning the entire table, so all the rows in the table will be read. The optimizer knows how many rows are in the table and does not need to estimate the number (see "Optimizer Accuracy" on page 4-27).

When an index is available, the optimizer must estimate the number of rows that will be read.

The optimizer will make the most accurate cost estimates about whether to use an index to access the data in cases when the search values are known. When the exact start and stop conditions are known at compilation time, the optimizer makes a very precise estimate of the number of rows that will be read.

If the index is unique, and the WHERE clause involves an = or IS NULL comparison to all the columns in the index, the optimizer knows that only a single row will be read.

In other circumstances, such as in the case when the statement's WHERE clause involves dynamic parameters that are known only at execution time and not at compilation time, or when the statement involves a join, the optimizer has to make a rougher estimate of the number of rows that will be read.

For example, the optimizer will make an accurate estimate of the cost of the following statement:

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

because the search value, 'SFO', is known. The optimizer will be able to make an accurate estimate of the cost of using the index *orig_index*. As explained in the section "When a Table Scan Is Better" on page 4-10, it may not be the best plan if the value 'SFO' appears in a large proportion of column *orig_airport*.

Stored prepared statements created or recompiled with sample values also allow the optimizer's decision to be well informed.

On the other hand, in the following statements, the search values are not known in advance:

```
-- dynamic parameters
SELECT *
FROM Flights
WHERE orig_airport = ?

-- joins
SELECT * FROM Countries, Cities
WHERE Countries.country_ISO_code = Cities.country_ISO_code

-- complex search conditions
SELECT * FROM Groups
WHERE tour_level = Tour->ECONOMYTOURLEVEL
```

In the above SELECT statements, the optimizer's decision as to whether to use indexes is less well informed.

Estimating Row Counts for Unknown Search Values

The way the optimizer estimates the number of rows that will be read when search values are unknown in advance is as follows: It multiplies the number of rows in the table by the *selectivity* for a particular operation. An operation's selectivity is a fixed number that attempts to describe the percentage of rows that will probably be returned; it may not correspond to the actual selectivity of the operation in every case. It is an assumption hard-wired into the Cloudscape system. For example, if a

particular operation has a selectivity of .5 (50%) and there are 100 rows in the index, the optimizer estimates that it will have to touch 50 rows. It uses this estimate in its cost estimate.

Table 4-1 shows what selectivity is assumed for various operations.

Table 4-1 Selectivity for Various Operations for Index Scans When Search Values Are Unknown in Advance

Operator	Selectivity
=, >=, >, <=, <, <> when data type of parameter is a boolean	.5 (50%)
other operators (except for IS NULL and IS NOT NULL) when data type of parameter is boolean	.5 (50%)
IS NULL	.1 (10%)
IS NOT NULL	.9 (90%)
=	.1 (10%)
>, >=, <, <=	.33 (3%)
<> compared to non-boolean type	.9 (90%)
LIKE transformed from LIKE predicate (see "LIKE Transformations" on page A-3)	1.0 (100%)
>= and < when transformed internally from LIKE (see "LIKE Transformations" on page A-3)	.25 (.5 X .5)
>= and <= operators when transformed internally from BETWEEN (see "BETWEEN Transformations" on page A-3)	.25 (.5 X .5)

About the Optimizer's Choice of Join Order

The optimizer chooses the optimal join order as well as the optimal index for each table. The join order can affect which index is the best choice. The optimizer may choose an index as the access path for a table if it is the inner table, but not if it is the outer table (and there are no further qualifications).

The optimizer chooses the join order of tables only in simple FROM clauses. Most joins using the JOIN keyword are flattened into simple joins, so the optimizer chooses their join order.

The optimizer does not choose the join order for outer joins; it uses the order specified in the statement.

When selecting a join order, the optimizer takes into account:

- the size of each table

- the indexes available on each table
- whether an index on a table is useful in a particular join order
- the number of rows and pages to be scanned for each table in each join order

NOTE: Cloudscape does transitive closure on qualifications. For details, see "Transitive Closure" on page A-7.

Join Order Case Study

For example, consider the following situation:

The *Flights* table (as you know) stores information about flight segments. It has a primary key on the *flight_id* and *segment_number* columns. This primary key constraint is backed up by a unique index on those columns.

The *FlightAvailability* table, which stores information about the availability of flight segments on particular days, may store several rows for a particular row in the *Flights* table (one for each date).

You want to see information about all the flights, and you issue the following query:

```
SELECT *
FROM FlightAvailability AS fa, Flights AS fts
WHERE fa.flight_id = fts.flight_id
AND fa.segment_number = fts.segment_number
```

First imagine the situation in which there are no useful indexes on the *FlightAvailability* table.

Using the join order with *FlightAvailability* as the outer table and *Flights* as the inner table is cheaper because it allows the *flight_id*/*segment_number* columns from *FlightAvailability* to be used to probe into and find matching rows in *Flights*, using the primary key index on *Flights.flight_id* and *Flights.segment_number.*

This is preferable to the opposite join order—with *Flights* as the outer table and *FlightAvailability* as the inner table—because in that case, for each row in *Flights*, the system would have to scan the entire *FlightAvailability* table to find the matching rows (since there is no useful index—one on the *flight_id*/*segment_number* columns).

Second, imagine the situation in which there is a useful index on the *FlightAvailability* table (this is actually the case in the sample database). *FlightAvailability* has a primary key index on *flight_id*, *segment_number*, and *booking_date*. In that index, the *flight_id-segment_number* combination is not unique, since there is a one-to-many correspondence between the *Flights* table and

the *FlightAvailability* table. However, the index is still very useful for finding rows with particular *flight_id*/*segment_number* values.

You issue the same query:

```
SELECT *
FROM FlightAvailability AS fa, Flights AS fts
WHERE fa.flight_id = fts.flight_id
AND fa.segment_number = fts.segment_number
```

Although the difference in cost is smaller, it is still cheaper for the *Flights* table to be the inner table, because its index is unique, whereas *FlightAvailability*'s index is not. That is because it is cheaper for Cloudscape to step through a unique index than through a non-unique index.

About the Optimizer's Choice of Join Strategy

The optimizer compares the cost of choosing a hash join (if a hash join is possible) to the cost of choosing a nested loop join and chooses the cheaper strategy. For information about when hash joins are possible, see "Join Strategies" on page 4-13.

In some cases, the size of the hash table that Cloudscape would have to build is prohibitive and can cause the JVM to run out of memory. For this reason, the optimizer has an upper limit on the size of a table on which it will consider a hash join. It will not consider a hash join for a statement if it estimates that the size of the hash table would exceed 1 MB. The optimizer's estimates of size of hash tables are approximate only.

NOTE: This limit does not apply if the query explicitly specifies hash join in an optimizer override (see "Overriding the Optimizer's Choice of Join Strategy" on page 4-26).

About the Optimizer's Choice of Sort Avoidance

Some SQL statements require that data be ordered, including those with ORDER BY, GROUP BY, and DISTINCT. MIN() and MAX() aggregates also require ordering of data.

Cloudscape can sometime avoid sorting steps for:

- statements with ORDER BY

  See "Cost-Based ORDER BY Sort Avoidance" on page 4-19

Cloudscape can also perform the following optimizations, but they are not based on cost:

- sort avoidance for DISTINCT and GROUP BYs
  See “Non-Cost-Based Sort Avoidance (Tuple Filtering)” on page 4-30
- statements with a MIN() aggregate
  See “The MIN() and MAX() Optimizations” on page 4-32

Cost-Based ORDER BY Sort Avoidance

Usually, sorting requires an extra step to put the data into the right order. This extra step can be avoided for data that are already in the right order. For example, if a single-table query has an ORDER BY on a single column, and there is an index on that column, sorting can be avoided if Cloudscape uses the index as the access path.

Where possible, Cloudscape’s query compiler transforms an SQL-J statement internally into one that avoids this extra step. For information about internal transformations, see “Sort Avoidance” on page A-17. This transformation, if it occurs, happens before optimization. After any such transformations are made, the optimizer can do its part to help avoid a separate sorting step by choosing an already sorted access path. It compares the cost of using that path with the cost of sorting. In Version 3.0, it does this for statements that use an ORDER BY clause in the following situations:

- The statements involve tables with indexes that are in the correct order.
- The statements involve scans of unique indexes that are guaranteed to return only one row per scan.

ORDER BY specifies a priority of ordering of columns in a result set. For example, ORDER BY X, Y means that column *X* has a more significant ordering than column *Y*.

The situations that allow Cloudscape to avoid a separate ordering step for statements with ORDER BY clauses are:

- Index scans, which provide the correct order.

```
-- covering index
SELECT flight_id FROM Flights ORDER BY flight_id
```

- The rows from a table when fetched through an index scan.

```
-- if Cloudscape uses the index on orig_airport
-- to access the data, it can avoid the sort
-- required by the final ORDER BY
SELECT orig_airport, miles
FROM FLIGHTS
```

```
WHERE orig_airport < 'DDD'
ORDER BY orig_airport
```

- The rows from a join when ordered by the indexed column or columns in the outer table.

```
-- if Cloudscape chooses Cities as the outer table, it
-- can avoid a separate sorting step
SELECT * FROM cities, countries
WHERE cities.country_ISO_code = countries.country_ISO_code
AND cities.country_ISO_code < 'DD'
ORDER BY cities.country_ISO_code
```

- Result sets that are guaranteed to return a single row. They are ordered on *all* of their columns (for example, if there are equality conditions on all the columns in a unique index, all the columns returned for that table can be considered ordered, with any priority of ordering of the columns).

```
-- query will only return one row, so that row is
-- "in order" for ANY column
SELECT miles
FROM Flights
WHERE flight_id = 'US1381' AND segment_number = 2
ORDER BY miles
```

- Any column in a result set that has an equality comparison with a constant. The column is considered ordered with no priority to its ordering.

```
-- The comparison of segment_number
-- to a constant means that it is always correctly
-- ordered. Using the index on (flight_id, segment_number)
-- as the access path means
-- that the ordering will be correct for the ORDER BY
-- clause in this query. The same thing would be true if
-- flight_id were compared to a constant instead.
SELECT segment_number, flight_id
FROM Flights
WHERE segment_number=2
ORDER BY segment_number, flight_id
```

  And because of transitive closure, this means that even more complex statements can avoid sorting. For example:

```
-- transitive closure means that Cloudscape will
-- add this clause:
-- AND countries.country_ISO_code = 'CL', which means
-- that the ordering column is now compared to a constant,
-- and sorting can be avoided.
```

```
SELECT * FROM cities, countries
WHERE cities.country_ISO_code = 'CL'
AND cities.country_ISO_code = countries.country_ISO_code
ORDER BY countries.country_ISO_code
```

For more information about transitive closure and other statement transformations, see Appendix A, "Internal Language Transformations".

- Simple Values clauses. Simple values clauses are flattened, allowing ordering on their columns to be eliminated. For example:

```
SELECT *
FROM (VALUES (1,2,3)) AS S (x,y,z),
(VALUES (1, 5, 6)) AS T(a,b,c)
WHERE s.x = t.a
ORDER BY s.x
```

In this case, the values clauses is flattened so that *s.x* is known to be a constant, and sorting can be avoided.

About the Optimizer's Selection of Lock Granularity

When a system is configured for row-level locking, the optimizer decides whether to use table-level locking or row-level locking for each table in each DML statement. The optimizer bases this decision on the number of rows read or written for each table, and on whether a full conglomerate scan is done for each table.

NOTE: When you have turned off row-level locking for your system, Cloudscape always uses table-level locking. The optimizer's decisions are ignored.

The first goal of the optimizer's decision is concurrency; wherever possible, the optimizer chooses row-level locking. However, row-level locking uses a lot of resources and may have a negative impact on performance. Sometimes row-level locking does not provide much more concurrency than table-level locking. In those situations, the optimizer may choose to escalate the locking scheme from row-level locking to table-level locking to improve performance. For example, if a connection is configured for TRANSACTION_SERIALIZABLE isolation, the optimizer chooses table-level locking for the following statement:

```
SELECT *
FROM FlightAvailability AS fa, Flights AS fts
WHERE fts.flight_id = fa.flight_id
AND fts.segment_number = fa.segment_number
```

To satisfy the isolation requirements, Cloudscape would have to lock all the rows in both the *FlightAvailability* and the *Flights* tables. Locking both the tables would

be cheaper, would provide the same isolation, and would allow the same concurrency.

NEW: Beginning in Version 3.0, you can force lock escalation for specific tables when you create or alter them with a SET LOCKING clause. For these tables, Cloudscape always chooses table-level locking. For more information, see "CREATE TABLE statement" on page 1-48 in the *Cloudscape Reference Manual*.

How the Optimizer Makes Its Decision at Compile Time

The optimizer determines the statement execution plan when a statement is compiled. A statement's isolation level may change at execution time; the optimizer cannot know what the statement's runtime isolation level will be. Therefore, the optimizer makes a decision about the lock granularity (whether to lock rows or entire tables) for a statement assuming that the statement will have TRANSACTION_SERIALIZABLE isolation, since that is the most restrictive. The optimizer uses the following rules:

- If the statement scans the entire table or index, the optimizer chooses table-level locking. (A statement scans the entire table whenever it chooses a table as the access path.)
- If a statement partially scans the index, the optimizer makes its decision based on its estimate of the number of index rows that will be *touched* during statement execution. If the estimated number of rows touched exceeds a threshold number, the optimizer chooses table-level locking. (You can configure this threshold number; see "Lock Escalation Threshold" on page 4-24.)
  - For SELECT, UPDATE, and DELETE statements, the number of rows touched is different from the number of rows read. If the same row is read more than once, it is considered to have been touched only once. Each row in the inner table of a join may be read many times, but may be touched at most one time.
  - For INSERT statements, the number of rows touched in the table being inserted into will be same as the number of rows inserted.
- For INSERT statements with *insertMode* set to *bulkInsert* or *replace*, the optimizer always chooses table-level locking.

Here are some examples that illustrate the choices the optimizer makes at compile time:

```
-- Optimizer chooses table locking for full scan
SELECT *
FROM my_table

-- Small number of rows matched,
-- optimizer chooses row locking
SELECT *
FROM my_table
WHERE primary_key = 17

-- Large number of rows matched,
-- optimizer chooses table locking
SELECT *
FROM my_table
WHERE sequence_number >= 1
AND sequence_number <= 10000

-- Presume other_table is inner table
-- optimizer will choose table locking
-- due to large number of rows touched
SELECT *
FROM million_row_table, other_table
WHERE million_row_table.x = other_table.primary_key

-- Row locking, because only one row inserted
INSERT INTO my_table VALUES (10, 12, 3)

-- Table locking on my_table, because of
-- large number of rows inserted
INSERT INTO my_table SELECT p FROM million_row_table

-- Table locking on my_table because of bulk insert
INSERT INTO my_table PROPERTIES insertMode = bulkInsert
SELECT *
FROM NEW FileImport('myfile.asc') AS t
-- Row locking due to small number of rows updated
UPDATE my_table SET x = 3
WHERE primary_key = 17
--Table locking due to all rows affected (table scan)
UPDATE my_table SET a = 3
-- Table locking due to large number of rows deleted
DELETE FROM my_table
WHERE sequence_number >= 1
AND sequence_number <= 10000
```

Lock Escalation Threshold

The system property *cloudscape.locks.escalationThreshold* determines the threshold for number of rows touched for a particular table above which the optimizer will choose table-level locking. The default value of this property is 5000. For large systems, set this property to a higher value. For smaller systems, lower it.

This property also sets the threshold for transaction-based lock escalation (see "Transaction-Based Lock Escalation" on page 4-28).

Runtime Overrides

The optimizer's assumptions about isolation level would diminish the benefits of having a lower isolation level if its decisions could not be overridden. Before executing statements, the runtime system makes the following override:

- For SELECT statements running in READ_COMMITTED isolation, the system always chooses row-level locking.

For all other statements (which require exclusive locks), the system lets the optimizer's compile-time decisions stand.

NOTE: For more information about lock escalation, see "Locking and Performance" on page 4-28.

About the Optimizer's Selection of Bulk Fetch

When Cloudscape retrieves data from a conglomerate, it can fetch more than one row at a time. Fetching more than one row at a time is called bulk fetch. By default, Cloudscape fetches 16 rows at a time.

Bulk fetch is faster than retrieving one row at a time when a large number of rows qualify for each scan of the table or index. Bulk fetch uses extra memory to hold the prefetched rows, so it should be avoided in situations in which memory is scarce.

Bulk fetch is automatically turned off for updatable cursors, for hash joins, for statements in which the scan returns a single row, and for subqueries. It is useful, however, for table scans or index range scans:

```
SELECT *
FROM Flights
WHERE miles > 4

SELECT *
FROM Flights
```

The default size for bulk fetch (16 rows) typically provides good performance.

Overriding the Optimizer

The optimizer usually makes a good choice about access path and join order. In addition, the default join strategy (nested loop) and the default bulk fetch buffer size (16) are usually appropriate. However, in some complex situations, the optimizer may not make the best choice, and the defaults may not be optimal. Cloudscape allows you to add various PROPERTIES clauses to an SQL-J statement to force a particular choice or to choose a value other than the default.

You can override the optimizer's choice of:

- access path
- join order
- join strategy (new in 2.0)
- join type
- bulk fetch size

In addition, you can tune the ways the optimizer and the system escalate locking to improve performance.

To override the optimizer, use an optimizer override property in the SQL-J statement's FROM clause or after a particular *tableExpression*.

See Chapter 6, "Optimizer Overrides", for complete details on optimizer override properties.

Overriding the Optimizer's Choice of Access Path

You may want to force a particular access path such as a table scan. For example:

```
SELECT *
FROM Countries PROPERTIES index=null
WHERE country_ISO_code = 'US'
```

See the *index* and *constraint* properties in Chapter 6, "Optimizer Overrides".

Overriding the Optimizer's Choice of Join Order

The *joinOrder* property in the FROM clause allows you to "fix" a particular join order—the order of items as they appear in the FROM clause. Otherwise, the optimizer makes its own choice about join order. For example:

```
SELECT Country, City
FROM PROPERTIES joinOrder=FIXED Countries, Cities
WHERE Countries.country_ISO_code = Cities.country_ISO_code
```

See the *joinOrder* property in Chapter 6, “Optimizer Overrides”.

Overriding the Optimizer’s Choice of Join Strategy

Cloudscape allows you to override the optimizer’s choice of join strategy in a PROPERTIES clause. In most cases, the optimizer makes a good choice of join strategy. For example:

```
SELECT *
FROM PROPERTIES joinOrder = FIXED
    FlightAvailability AS fa, Flights AS fts
PROPERTIES joinStrategy=hash
WHERE fts.flight_id = fa.flight_id
AND fts.segment_number = fa.segment_number
```

See the *joinStrategy* property in Chapter 6, “Optimizer Overrides”.

Overriding the Optimizer’s Choice of Fetch Size

Cloudscape allows you to override the default bulk-fetch buffer size (16) in a PROPERTIES clause. In future releases, the optimizer will automatically choose the best bulk-fetch size for each table in a query.

For example:

```
SELECT *
FROM Flights
PROPERTIES bulkFetch = 64
```

See the *bulkFetch* property in Chapter 6, “Optimizer Overrides”.

Tuning Lock Granularity

As explained in the section “About the Optimizer’s Selection of Lock Granularity” on page 4-21, you can tune the ways both the optimizer and the runtime system escalate locks from row-level locking to table-level locking to improve performance. See also “*cloudscape.locks.escalationThreshold*” on page 5-48.

Optimizer Accuracy

The optimizer estimates are not guaranteed to be accurate. The optimizer uses a stored row count to determine the number of rows in a table, which is maintained automatically by the system. However, it is not stored for tables that were created prior to Version 1.5 and upgraded. This could cause the optimizer to come up with suboptimal plans for these tables.

Normally, an updated value is stored in the database whenever the database goes through an orderly shutdown (as long as the database is not read-only). Stored row counts become inaccurate if there is a non-orderly shutdown (for example, a power failure or other type of system crash).

You can correct the optimizer's row count without shutting down the system; Cloudscape sets the stored row count for a table to the correct value whenever a query that does a full scan on the base conglomerate finishes. For example, executing the following query sets the row count for table *Flights* to the correct value:

```
SELECT *
FROM Flights PROPERTIES index=null
```

Cloudscape also sets the stored row count on a table to the correct value whenever a user creates a new index or primary key, unique, or foreign key constraint on the table. This value is not guaranteed to be written to disk.

Providing Costing Information for VTIs

Cloudscape allows you to make costing information available to the optimizer with some optional extensions.

NEW: Costing information for a VTI class is new in Version 3.0. In addition, you can specify that a VTI class can only be instantiated once during the compilation and execution of a statement. Finally, another change is that a VTI can be chosen for a hash join.

By implementing the optional interface *COM.cloudscape.vti.VTICosting*, your VTI class can provide the following information to the Cloudscape optimizer:

- estimated cost
- estimated cost per instantiation
- whether the class supports multiple instantiations

  If the class does not support multiple instantiations, and the external virtual table instantiation of it is used in the context of a join, Cloudscape will

consider the external virtual table as the inner table only if the join is a hash join or a nested loop join in which the inner table is materialized. (In such a situation, if the external virtual table instantiation of the class takes a join column as a parameter, the optimizer cannot choose a legal join order for the query and an exception is thrown.)

For information about programming VTI classes to provide such information, see "Providing Costing Information" on page 5-22 in the *Cloudscape Developer's Guide*.

Locking and Performance

Row-level locking improves concurrency in a multi-user system. However, a large number of row locks can degrade performance. "About the Optimizer's Selection of Lock Granularity" on page 4-21 discussed the way the optimizer makes some compile-time decisions about escalating row locks to table locks for performance reasons. This section discusses ways in which the Cloudscape system and the user can make similar lock escalations.

- "Transaction-Based Lock Escalation" on page 4-28
- "LOCK TABLE Statement" on page 4-30

Transaction-Based Lock Escalation

The optimizer makes its decisions for the scope of a single statement at compile time; the runtime overrides are also for the scope of a single statement. As you know, a transaction may span several statements. For connections running in TRANSACTION_SERIALIZABLE isolation and for connections that are doing a lot of inserts or updates, a transaction may accumulate a number of row locks even though no single statement would touch enough rows to make the optimizer choose table-level locking for any single table.

However, during a transaction, the Cloudscape system tracks the number of locks for all tables in the transaction, and when this number exceeds a threshold number (which you can configure; see "Lock Escalation Threshold" on page 4-24), the system attempts to escalate locking for at least one of the tables involved from row-level to table-level locking.

The system attempts to escalate to table-level locking for each table that has a burdensome number of locks by trying to obtain the relevant table lock. If the

system can lock the table without waiting, the system locks the entire table and releases all row locks for the table. If the system cannot lock the table without waiting, the system leaves the row locks intact.

Once a table is locked in either mode, a transaction does not acquire any subsequent row-level locks on a table. For example, if a transaction locks the entire *Hotels* table in share mode in order to read data, it may later need to lock a particular row in exclusive mode in order to update the row. However, the previous table-level lock on *Hotels* forces the exclusive lock to be table-level as well.

This transaction-based runtime decision is independent of any compilation decision.

If when the escalation threshold was exceeded the system did not obtain any table locks because it would have had to wait, the next lock escalation attempt is delayed until the number of held locks has increased by some significant amount, for example from 5000 to 6000.

Here are some examples assuming the escalation threshold is 5000:

- single table holding the majority of the locks

Table	Number of Row Locks	Promote?
Hotels	4853	yes
Countries	3	no
Cities	12	no

- two tables holding the majority of the locks

Table	Number of Row Locks	Promote?
Hotels	2349	yes
Countries	3	no
Cities	1800	yes

- many tables holding a small number of locks

Table	Number of Row Locks	Promote?
table001	279	no
table002	142	no
table003	356	no
table004	79	no

Table	Number of Row Locks	Promote?
table194	384	no
table195	416	no

LOCK TABLE Statement

In addition, you can explicitly lock a table for the duration of a transaction with the LOCK TABLE statement. This is useful if you know in advance that an entire table should be locked and want to save the resources required for obtaining row locks until the system escalates the locking. For information about this feature, see "LOCK TABLE statement" on page 1-91 of the *Cloudscape Reference Manual*.

Non-Cost-Based Optimizations

The optimizer makes some non-cost-based optimizations, which means that it does not consider them when determining the access path and join order. If all the conditions are right, it makes the optimizations after the access path and join order are determined.

- "Non-Cost-Based Sort Avoidance (Tuple Filtering)" on page 4-30
- "The MIN() and MAX() Optimizations" on page 4-32

Non-Cost-Based Sort Avoidance (Tuple Filtering)

In most cases, Cloudscape needs to perform two separate steps for statements that use DISTINCT or GROUP BY: first sorting the selected columns, then either discarding duplicate rows or aggregating grouped rows. Sometimes it is able to avoid sorting for these statements with tuple filtering. *Tuple filtering* means that the rows are *already* in a useful order. For DISTINCT, Cloudscape can simply filter out duplicate values when they are found and return results to the user sooner. For GROUP BY, Cloudscape can aggregate a group of rows until a new set of rows is detected and return results to the user sooner.

These are non-cost-based optimizations; the optimizer does not yet consider the cost of these optimizations.

The examples in this section refer to the following tables:

```
CREATE TABLE t1(c1 INT, c2 INT, c3 INT, c4 INT)

CREATE INDEX i1 ON t1(c1)

CREATE INDEX i1_2_3_4 ON t1(c1, c2, c3, c4)
```

DISTINCT

Tuple filtering is applied for a DISTINCT when the following criteria are met:

- The SELECT list is composed entirely of simple column references and constants.
- All simple column references come from the same table and the optimizer has chosen the table in question to be the outermost table in the query block.
- The optimizer has chosen an index as the access path for the table in question.
- The simple column references in the SELECT list, plus any simple column references from the table that have equality predicates on them, are a prefix of the index that the optimizer selected as the access path for the table.

NOTE: The set of column references must be an in-order prefix of the index.

Here is the most common case in which tuple filtering will be applied:

```
SELECT DISTINCT c1 FROM t1
```

Equality predicates allow tuple filtering on the following:

```
SELECT DISTINCT c2
FROM t1 PROPERTIES index = i1_2_3_4
WHERE c1 = 5

SELECT DISTINCT c2, c4
FROM t1 PROPERTIES index = i1_2_3_4
WHERE c1 = 5 and c3 = 7

-- the columns don't have to be in the
-- same order as the index
SELECT DISTINCT c2, c1
FROM t1 PROPERTIES index = i1_2_3_4
```

Quick DISTINCT Scans

Cloudscape can use a hash table instead of a sorter to eliminate duplicates when performing a DISTINCT in the following cases:

- There is a single table in the query block.
- An ORDER BY clause is not merged into the DISTINCT.
- All entries in the SELECT list are simple column references.
- There are no predicates in the query block.

This technique allows for minimal locking when performing the scan at the READ COMMITTED isolation level.

NEW: This technique appears in RunTimeStatistics as a *DistinctScanResultSe*t and is new in Version 3.0.

GROUP BY

Tuple filtering is applied for a GROUP BY when the following criteria are met:

- All grouping columns come from the same table and the optimizer has chosen the table in question to be the outermost table in the query block.
- The optimizer has chosen an index as the access path for the table in question.
- The grouping columns, plus any simple column references from the table that have equality predicates on them, are a prefix of the index that the optimizer selected as the access path for the table.

Here is the most common case in which tuple filtering will be applied:

```
SELECT max(c2) FROM t1 GROUP BY c1
```

Equality predicates allow tuple filtering on the following:

```
SELECT c2, SUM(c3)
FROM t1 PROPERTIES index = i1_2_3_4
WHERE c1 = 5 GROUP BY c2

SELECT max(c4)
FROM t1 PROPERTIES index = i1_2_3_4
WHERE c1 = 5 AND c3 = 6 GROUP BY c2
```

The MIN() and MAX() Optimizations

The optimizer knows that it can avoid iterating through all the source rows in a result to compute a MIN() or MAX() aggregate when data are already in the right order. When data are guaranteed to be in the right order, Cloudscape can go immediately to the smallest (minimum) or largest (maximum) row.

The following conditions must be true:

- The MIN() or MAX() is the only entry in the SELECT list.
- The MIN() or MAX() is on a simple column reference, not on an expression.
- For MAX(), there must not be a WHERE clause.
- For MIN():
  - The referenced table is the outermost table in the optimizer's chosen join order for the query block.
  - The optimizer chose an index containing the referenced column as the access path.
  - The referenced column is the first key column in that index OR the referenced column is a key column in that index and equality predicates exist on all key columns prior to the simple column reference in that index.

NEW: The MAX() optimization is new in Version 3.0.

For example, the optimizer can use this optimization for the following queries (if the optimizer uses the appropriate indexes as the access paths):

```
-- index on orig_airport
SELECT MIN(orig_airport)
FROM Flights

-- index on orig_airport
SELECT MAX(orig_airport)
FROM Flights

-- index on orig_airport
SELECT miles FROM Flights
WHERE orig_airport = (SELECT MIN(orig_airport)
FROM Flights)

-- index on segment_number, flight_id
SELECT MIN(segment_number) FROM Flights
WHERE flight_id = 'AA1111'

SELECT * FROM Flights WHERE segment_number = (SELECT
MIN(segment_number) FROM Flights
WHERE flight_id = 'AA1111')
```

The optimizer decides whether to implement the optimization after choosing the plan for the query. The optimizer does not take this optimization into account when costing the plan.

5 Cloudscape Properties

A property in Cloudscape belongs to one or more of these scopes (for more information about scopes, precedence, and persistence, see “Properties Overview” on page 1-1):

- *system-wide*
  System-wide properties apply to an entire system, including all its databases and tables if applicable.

  - Set programmatically

  System-wide properties set programmatically have precedence over database-wide properties and system-wide properties set in the *cloudscape.properties* file.

  - Set in the *cloudscape.properties* file
- *database-wide*
  A database-wide property is stored in a database and is valid for that specific database only.

  To be included in a publication in a synchronization system, a property must be a database-wide property. You can publish any database-wide property. Full-access users can also set database properties at targets.

NOTE: Database-wide properties are stored in the database and are simpler for deployment. System-wide parameters are probably easier for development.

- *conglomerate-specific*
  Conglomerate-specific properties are set at table or index creation time and define storage attributes for the specific table or index only; the rest of the database uses the value of the property that is set at a database-wide or system-wide level. For more information, see “Conglomerate-Specific Properties” on page 1-8.

This chapter includes all the core Cloudscape properties. For synchronization-related properties, see the *Cloudscape Synchronization Guide*. For Cloudconnector properties (which are set in a different file), see the *Cloudscape Server and Administration Guide*. For optimizer override properties, see Chapter 6, "Optimizer Overrides".

> **NOTE:** When setting properties that have boolean values, be sure to trim extra spaces around the word *true*. Extra spaces around the word *true* cause the property to be set to false.

Table 5-1 summarizes the general Cloudscape properties. In that table, S stands for system-wide, and D stands for database-wide. X means yes.

Table 5-1 Cloudscape Properties

Property	New in 3.0	Scope	Publishable	Dynamic
"*cloudscape.authentication.ldap.searchAuthDN*" on page 5-5		S, D	X	
"*cloudscape.authentication.ldap.searchAuthPW*" on page 5-7		S, D	X	
"*cloudscape.authentication.ldap.searchBase*" on page 5-8		S, D	X	
"*cloudscape.authentication.ldap.searchFilter*" on page 5-9		S, D	X	
"*cloudscape.authentication.provider*" on page 5-11		S, D	X	
"*cloudscape.authentication.server*" on page 5-13		S, D	X	
"*cloudscape.connection.requireAuthentication*" on page 5-15		S, D	X	
"*cloudscape.database.classpath*" on page 5-17		S, D	X	X*
"*cloudscape.database.defaultConnectionMode*" on page 5-19		S, D	X	X*
"*cloudscape.database.forceDatabaseLock*" on page 5-21	X	S		
"*cloudscape.database.fullAccessUsers*" on page 5-22		S, D	X	X*
"*cloudscape.database.noAutoBoot*" on page 5-24		D		
"*cloudscape.database.propertiesOnly*" on page 5-25		D	X	X
"*cloudscape.database.readOnlyAccessUsers*" on page 5-26		S, D	X	X*
"*cloudscape.infolog.append*" on page 5-28		S		

Table 5-1 Cloudscape Properties

Property	New in 3.0	Scope	Publishable	Dynamic
"*cloudscape.jdbc.metadataStoredPreparedStatements*" on page 5-29		S		X
"*cloudscape.language.bulkFetchDefault*" on page 5-31		S, D	X	X
"*cloudscape.language.defaultIsolationLevel*" on page 5-33		S, D	X	
"*cloudscape.language.logStatementText*" on page 5-35		S, D	X	
"*cloudscape.language.preloadClasses*" on page 5-36		S, D	X	X
"*cloudscape.language.spsCacheSize*" on page 5-37		S, D	X	
"*cloudscape.language.stalePlanCheckInterval*" on page 5-39	X	S, D	X	
"*cloudscape.language.statementCacheSize*" on page 5-41		S, D	X	
"*cloudscape.language.triggerMaximumRecursionLevel*" on page 5-43	X	D	X	X
"*cloudscape.locks.deadlockTimeout*" on page 5-45		S, D	X	X
"*cloudscape.locks.deadlockTrace*" on page 5-47	X	S, D	X	
"*cloudscape.locks.escalationThreshold*" on page 5-48		S, D	X	X
"*cloudscape.locks.deadlockTrace*" on page 5-47	X	S, D	X	X
"*cloudscape.locks.monitor*" on page 5-50	X	S, D	X	X
"*cloudscape.locks.waitTimeout*" on page 5-51	X	S,D	X	X
"*cloudscape.service*" on page 5-53		S		X
"*cloudscape.storage.initialPages*" on page 5-55		C		
"*cloudscape.storage.minimumRecordSize*" on page 5-57		S, D, C	X	X
"cloudscape.storage.pageCacheSize" on page 5-59		S		
"*cloudscape.storage.pageReservedSpace*" on page 5-60		S, D, C	X	X
"*cloudscape.storage.pageSize*" on page 5-62		S, D, C	X	X
"*cloudscape.storage.rowLocking*" on page 5-64		S, D	X	
"*cloudscape.storage.tempDirectory*" on page 5-66		S, D		X
"*cloudscape.stream.error.field*" on page 5-68		S		
"*cloudscape.stream.error.file*" on page 5-69		S		
"*cloudscape.stream.error.logSeverityLevel*" on page 5-70		S		

Table 5-1 Cloudscape Properties

Property	New in 3.0	Scope	Publishable	Dynamic
"*cloudscape.stream.error.method*" on page 5-71		S		
"*cloudscape.system.bootAll*" on page 5-72		S		
"*cloudscape.system.home*" on page 5-73		S		
"*cloudscape.user.UserName*" on page 5-74		S, D	X	X

*

See the man page for this property for information about when changes to it are dynamic.

cloudscape.authentication.ldap.searchAuthDN

Along with *cloudscape.authentication.ldap.searchAuthPW*, this property indicates how Cloudscape should bind with the LDAP directory server to do searches for user DN (distinguished name). This property specifies the DN; *cloudscape.authentication.ldap.searchAuthPW* specifies the password to use for the search.

If these two properties are not specified, an anonymous search is performed if it is supported.

For more information about LDAP user authentication, see "LDAP Directory Service" on page 8-9 of the *Cloudscape Developer's Guide.*

Syntax

```
cloudscape.authentication.ldap.searchAuthDn=DN
```

Default

If not specified, an anonymous search is performed if it is supported.

Example

```
-- system-wide property
cloudscape.authentication.ldap.searchAuthDn=
    cn=guest,o=cloudscape.com

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.authentication.ldap.searchAuthDn',
    'cn=guest,o=cloudscape.com')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.authentication.ldap.searchAuthDn=
    'cn=guest,o=cloudscape.com'
```

Scope

system-wide

database-wide (publishable)

Static or Dynamic

Static. For system-wide properties, you must reboot Cloudscape for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

cloudscape.authentication.ldap.searchAuthPW

Along with *cloudscape.authentication.ldap.searchAuthDN*, indicates how Cloudscape should bind with the directory server to do searches in order to retrieve a fully qualified user DN. This property specifies the password; *cloudscape.authentication.ldap.searchAuthDN* specifies the DN to use for the search.

For more information about LDAP user authentication, see “LDAP Directory Service” on page 8-9 of the *Cloudscape Developer’s Guide.*

Default

If not specified, an anonymous search is performed if it is supported.

Example

```
-- system-wide property
cloudscape.authentication.ldap.searchAuthPW=guestPassword

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.authentication.ldap.searchAuthPW',
    'guestPassword')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.authentication.ldap.searchAuthPW=
    'guestPassword'
```

Scope

system-wide

database-wide (publishable)

Static or Dynamic

Static. For system-wide properties, you must reboot Cloudscape for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

cloudscape.authentication.ldap.searchBase

Function

Specifies the root DN of the point in your hierarchy from which to begin a guest or anonymous search for the user's DN. For example:

```
ou=people,o=cloudscape.com
```

When using Netscape Directory Server, set this property to the root DN, the special entry to which access control does not apply.

For more information about LDAP user authentication, see "LDAP Directory Service" on page 8-9 of the *Cloudscape Developer's Guide.*

Example

```
-- system-wide property
cloudscape.authentication.ldap.searchBase=
    ou=people,o=cloudscape.com

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.authentication.ldap.searchBase',
    'ou=people,o=cloudscape.com')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.authentication.ldap.searchBase=
    'ou=people,o=cloudscape.com'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Static. For system-wide properties, you must reboot Cloudscape for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

cloudscape.authentication.ldap.searchFilter

Function

Specifies the search filter to use to determine what constitutes a user (and other search predicate) for Cloudscape searches for a full DN during user authentication.

If you set this property to *cloudscape.user*, Cloudscape looks for cached full DNs for users that you have defined with the *cloudscape.user.UserName* property. For other users, Cloudscape performs a search using the *default* search filter.

For more information about LDAP user authentication, see "LDAP Directory Service" on page 8-9 of the *Cloudscape Developer's Guide.*

Syntax

```
cloudscape.authentication.ldap.searchFilter=
    { searchFilter | cloudscape.user)
```

Default

```
(&(objectClass=inetOrgPerson)(uid=userName))
```

NOTE: Cloudscape automatically &s the filter you specify with ((uid=*userName*)) unless you include %USERNAME% in the definition. You may want to use %USERNAME% if your user DNs map the user name to something other than *uid* (for example, *user*).

Examples

```
-- system-wide properties

cloudscape.authentication.ldap.searchFilter=objectClass=person
## people in the marketing department
## Cloudscape automatically adds (uid=<userName>)
cloudscape.authentication.ldap.searchFilter=(&(ou=Marketing)
    (objectClass=person))
## all people but those in marketing
## Cloudscape automatically adds (uid=<userName>)
cloudscape.authentication.ldap.searchFilter=(&(!(ou=Marketing)
    (objectClass=person))
## map %USERNAME% to user, not uid
cloudscape.authentication.ldap.searchFilter=(&((ou=People)
(user=%USERNAME%))
```

```
## cache user DNs locally and use the default for others
cloudscape.authentication.ldap.searchFilter=cloudscape.user

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.authentication.ldap.searchFilter',
    'objectClass=person')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.authentication.ldap.searchFilter=
    'objectClass=person'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Static. For system-wide properties, you must reboot Cloudscape for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

cloudscape.authentication.provider

Function

Specifies the authentication provider for Cloudscape user authentication.

Legal values include:

- LDAP

  An external LDAP directory service.
- NIS+

  An external NIS+ directory service.
- CLOUDSCAPE

  Cloudscape's simple internal user authentication repository.
- a complete Java class name

  A user-defined class that provides user authentication.

When using an external authentication service provider (LDAP or NIS+), you must also set:

- *cloudscape.authentication.server*

When using LDAP, you can set other LDAP-specific properties. See also:

- "*cloudscape.authentication.ldap.searchAuthDN*" on page 5-5
- "*cloudscape.authentication.ldap.searchAuthPW*" on page 5-7
- "*cloudscape.authentication.ldap.searchFilter*" on page 5-9
- "*cloudscape.authentication.ldap.searchBase*" on page 5-8

Alternatively, you can write your own class to provide a different external authentication service. This class must implement the public interface *COM.cloudscape.authentication.Interface.AuthenticationScheme* and throw exceptions of the type *COM.cloudscape.authentication.Interface.AuthenticationException* where appropriate. Using a user-defined class makes Cloudscape adaptable to various naming and directory services. For example, the class could allow Cloudscape to hook up to an existing user authentication service that uses any of the standard directory and naming service providers to JNDI.

To enable any Cloudscape user authentication, you must set the *cloudscape.connection.requireAuthentication* property to true.

For more information about user authentication, see "Working with User Authentication" on page 8-5 of the *Cloudscape Developer's Guide.*

Syntax

```
cloudscape.authentication.provider={
    LDAP |
    NIS+ |
    CLOUDSCAPE |
    classProviderName }
```

Example

```
-- system-wide property
cloudscape.authentication.provider=LDAP

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.authentication.provider',
    'CLOUDSCAPE')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.authentication.provider=
    'NIS+'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Static. For system-wide properties, you must reboot Cloudscape for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

cloudscape.authentication.server

Function

Specifies the location of the external directory service that provides user authentication for the Cloudscape system as defined with *cloudscape.authentication.provider*. For LDAP, specify the host name and port number. For NIS+, specify the NIS server name and the NIS domain name.

The server must be known on the network.

For more information about external user authentication, see “External Directory Service” on page 8-8 of the *Cloudscape Developer’s Guide.*

Default

Not applicable.

Syntax

```
cloudscape.authentication.server=
[{ ldap: | nisplus: }]
[//]

{
    hostname:portnumber |
    nisServerName/nisDomain
}
```

Example

```
-- system-wide property
##LDAP example
cloudscape.authentication.server=godfrey:9090
##LDAP example
cloudscape.authentication.server=ldap://godfrey:9090
##LDAP example
cloudscape.authentication.server=//godfrey:9090
##NIS+ example
cloudscape.authentication.server=//godfrey/cloudscape.com
##NIS+ example
cloudscape.authentication.server=nisplus://godfrey/
cloudscape.com
##NIS+ example
cloudscape.authentication.server=godfrey/cloudscape.com
```

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.authentication.server',
    'godfrey:9090')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.authentication.server=
    'godfrey:9090'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Static. For system-wide properties, you must reboot Cloudscape for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

cloudscape.connection.requireAuthentication

Function

Turns on user authentication for Cloudscape.

When user authentication is turned on, a connection request must provide a valid user name and password.

Cloudscape uses the type of user authentication specified with the *cloudscape.authentication.provider* property.

For more information about user authentication, see “Working with User Authentication” on page 8-5 of the *Cloudscape Developer’s Guide*.

Default

False.

By default, no user authentication is required.

Example

```
-- system-wide property
cloudscape.connection.requireAuthentication=true

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.connection.requireAuthentication',
    'true')

-- publishing a database-wide property
ALTER PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.connection.requireAuthentication=
    'true'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Static. For system-wide properties, you must reboot Cloudscape for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

cloudscape.database.classpath

Function

Enables class loading from the database. For a complete discussion of database class loading, see “Loading Classes from a Database” on page 3-17 of the *Cloudscape Developer’s Guide*.

Takes a list of fully qualified jar names that indicates which jar files the Cloudscape class loader should search for classes and other resources, and in which order. The search order is from left to right, so this property behaves like the standard class path.

The class loader looks first in the user’s class path, then in the jar files specified with this property. To ensure class loading from the database, remove classes from the user’s class path.

You can set this property for the following kinds of storage:

- Database storage. Classes are stored in the current database and are not available across databases. *Recommended*.

  Set it as a database-level property.

  The fully qualified name consists of a *two-part name*: schema name and an unqualified jar name.
- System storage. Classes are stored in the database you specify and are available across databases. *Not recommended* (but corresponds to Version 1.5 behavior and allows sharing across databases).

  Set it as a system-level property.

  The fully qualified name consists of a *three-part name*: database name, a schema name, and an unqualified jar name.

  Does not work when a database has user authentication turned on, because the system does not provide a user name and password when accessing the classes stored in the database.

NOTE: Do not mix two-part and three-part names. Such property values are invalid.

To specify multiple jar files, use a colon (:) as a separator.

Syntax

```
-- database-level, set only in an SQL-J statement
CALL PropertyInfo.setDatabaseProperty(
```

```
    'cloudscape.database.classpath', 'schemaName.
    unqualifiedJarFile[:schemaName.unqualifiedJarFile]*')
-- System-level
cloudscape.database.classpath=databaseName.schemaName.
unqualifiedJarFile[:databaseName.schemaName.unqualifiedJarFile]
*
```

Example

To set the search path for the current database to be from *APP.photo* and *APP."AccountinG"*:

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.classpath',
    'APP.photo:APP."AccountinG"')
```

To set the search path for Java classes to be from database *photodb*, schema *APP*, and jar file *photo* followed by database *general*, schema *APP,* and jar file *jgl*:

```
-- system-wide property
cloudscape.database.classpath=photodb.APP.photo:general.APP.jgl

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.database.classpath=
    'APP.photo:APP."AccountinG"'
```

Scope

system-wide (see notes above)

database-wide (see notes above) (publishable)

Dynamic or Static

For database storage, the first time this property is set, it is static; rebooting the database is required. If this property is set as a database property (and the value is valid), thereafter, the next time the property is set as a database-level property, no rebooting is required; the change is dynamic (see “Dynamic Changes to Jar Files or Database Jar Class Path” on page 3-26 of the *Cloudscape Developer’s Guide*). All other changes are static and require rebooting.

For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.database.defaultConnectionMode

Function

One of the user authorization properties.

Defines the default connection mode for users of the database or system for which this property is set. The possible values (which are case-insensitive) are:

- *noAccess*

  Disallows connections.
- *readOnlyAccess*

  Grants read-only connections.
- *fullAccess*

  Grants full access.

If the property is set to an invalid value, an exception is raised.

NOTE: It is possible to configure a database so that it cannot be changed (or even accessed) using this property. If you set this property to *noAccess* or *readOnlyAccess*, be sure to allow at least one user full access. See "*cloudscape.database.fullAccessUsers*" on page 5-22 and "*cloudscape.database.readOnlyAccessUsers*" on page 5-26.

For more information about user authorization, see "User Authorization" on page 8-23 of the *Cloudscape Developer's Guide*.

```
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.defaultConnectionMode',
    '{ noAccess | readOnlyAccess | fullAccess}')
```

Example

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.defaultConnectionMode', 'noAccess')

-- system-wide property
cloudscape.database.defaultConnectionMode=noAccess

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.database.defaultConnectionMode=
    'fullAccess'
```

Default

fullAccess

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Dynamic. Current connections are not affected, but all future connections are affected. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.database.forceDatabaseLock

NEW: The *cloudscape.database.forceDatabaseLock* property is new in Version 3.0.

Function

Cloudscape attempts to prevent two JVMs from accessing a database at one time (and potentially corrupting it) with the use of a file called *db.lck* in the database directory. On some operating systems, the use of a lock file does not guarantee single access, and so Cloudscape only issues a warning and may allow multiple JVM access even when the file is present. (For more information, see "Double-Booting System Behavior" on page 2-9 in the *Cloudscape Developer's Guide*.)

Cloudscape provides the property *cloudscape.database.forceDatabaseLock* for use on platforms that do not provide the ability for Cloudscape to guarantee single JVM access. When this property is set to true, if Cloudscape finds the *db.lck* file when it attempts to boot the database, it throws an exception and does not boot the database.

NOTE: This situation can occur even when no other JVMs are accessing the database; in that case, remove the *db.lck* file by hand in order to boot the database. If the *db.lck* file is removed by hand while a JVM is still accessing a Cloudscape database, there is no way for Cloudscape to prevent a second VM from starting up and possibly corrupting the database. In this situation no warning message is logged to the error log.

Example

```
cloudscape.database.forceDatabaseLock=true
```

Scope

system-wide

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.database.fullAccessUsers

Function

One of the user authorization properties. Specifies a list of users to which full (read-write) access to a database is granted. The list consists of user names separated by commas. Do not put spaces after commas.

When set as a system property, specifies a list of users for which full access to all the databases in the system is granted.

See also "*cloudscape.database.readOnlyAccessUsers*" on page 5-26.

A malformed list of user names raises an exception. Do not specify a user both with this property and in *cloudscape.database.readOnlyAccessUsers.*

NOTE: User names, called authorization identifiers, follow the rules of *SQL92Identifiers* and can be delimited. Specifying a user name that does not follow these rules raises an exception.

For more information about user authorization, see "User Authorization" on page 8-23 of the *Cloudscape Developer's Guide.*

Syntax

```
-- database-level property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.fullAccessUsers',
    'commaSeparatedlistOfUsers')
```

Example

```
-- database-level property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.fullAccessUsers', 'dba,fred,peter')

--system-level property
cloudscape.database.fullAccessUsers=dba,fred,peter

-- publishing a database-wide property
ALTER PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.database.fullAccessUsers=
    'dba,fred,peter'
```

Scope

database-wide (publishable)

system-wide

Dynamic or Static

Dynamic. Current connections are not affected, but all future connections are affected. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.database.noAutoBoot

Function

Specifies whether a database that is part of the current system is booted when Cloudscape starts up. If the *cloudscape.system.bootAll* property is set to true (which it is not by default), all databases in the current system are booted when Cloudscape starts up, unless the *cloudscape.database.noAutoBoot* property is set to true for a database. Setting this property to true for a database means that it is booted only upon the first connection.

NOTE: For information on what databases are considered part of the current system when it starts up, see "*cloudscape.service*" on page 5-53.

If *cloudscape.system.bootAll is* set to false, this property has no effect.

Default

False.

Example

```
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.noAutoBoot', 'true')
```

Scope

database-wide (not publishable)

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.database.propertiesOnly

Function

When set to true, this property ensures that database-wide properties cannot be overridden by system-wide properties.

When this property is set to false, or not set, database-wide properties can be overridden by system-wide properties (see “Precedence of Properties” on page 1-3).

This property ensures that a database’s environment cannot be modified by the environment in which it is booted. Typically most databases that are distributed or synchronization targets will require this property to be set to true.

This property can *never* be overridden by system properties.

Default

False.

Example

```
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.propertiesOnly','true')

-- publishing a database-wide property
ALTER PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.database.propertiesOnly=
    'true'
```

Scope

database-wide (publishable)

Dynamic or Static

This property is dynamic; if you change it while Cloudscape is running, the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.database.readOnlyAccessUsers

Function

One of the user authorization properties. Specifies a list of users to which read-only access to a database is granted. The list consists of user names separated by commas. Do not put spaces after commas.

When set as a system property, specifies a list of users for which read-only access to all the databases in the system is granted.

See also "*cloudscape.database.fullAccessUsers*" on page 5-22.

A malformed list of user names raises an exception. Do not specify a user both in this property and in *cloudscape.database.fullAccessUsers*.

NOTE: User names, called authorization identifiers, follow the rules of *SQL92Identifiers* and can be delimited. Specifying a user name that does not follow these rules raises an exception.

Syntax

```
-- database-level property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.readOnlyAccessUsers',
    'commaSeparatedistOfUsers')
```

Example

```
-- database-level property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.readOnlyAccessUsers', 'ralph,guest')

-- system-level property
cloudscape.database.readOnlyAccessUsers=ralph,guest

-- publishing a database-wide property
ALTER PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.database.readOnlyAccessUsers=
    'ralph,guest'
```

Scope

database-wide (publishable)

system-wide

Dynamic or Static

Dynamic. Current connection is not affected, but all future connections are affected. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.infolog.append

Function

Specifies whether to append to or destroy and re-create the *cloudscape.LOG* file, which is used to record error and other information when Cloudscape starts up in a JVM.

You can set this property even if the file does not yet exist; Cloudscape creates the file.

Default

False.

By default, the file is deleted and then re-created.

Example

```
cloudscape.infolog.append=true
```

Scope

system-wide

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.jdbc.metadataStoredPreparedStatements

Function

Cloudscape's local JDBC driver has built-in queries for the JDBC *DatabaseMetaData* methods that supply information about the Cloudscape system. These methods are useful for applications working with generic DBMSs, such as database tools.

The *cloudscape.jdbc.metadataStoredPreparedStatements* property configures the JDBC driver to take advantage of stored prepared statements to avoid preparing these queries each time a system starts up.

When configured to use stored prepared statements, the JDBC driver stores all JDBC metadata stored prepared statements, which are placed in the *SYS* schema. JDBC metadata queries are duplicated in every Cloudscape database even if a single Cloudscape instance manages several databases.

The full set of statements adds about 60K to the size of a database.

You can delete the stored prepared statements in a database created for these queries by executing the method *dropAllJDBCMetaDataSPSes* in the class *COM.cloudscape.database.Database* in an SQL-J statement. To create an instance of that class against which you can call the method, call *getDatabaseOfConnection()* in the class *COM.cloudscape.database.Factory* (aliased as *Factory*). For example:

```
CALL Factory.getDatabaseOfConnection().
    dropAllJDBCMetaDataSPSes()
```

The JDBC driver is responsible for the creation and retrieval of the JDBC metadata statements.

Possible Values (case-insensitive)

- *off*
  The JDBC driver does not create any stored prepared statements.
- *dynamic (DEFAULT)*
  The JDBC driver creates stored prepared statements on an as-needed basis. Each time a JDBC metadata request is made to the driver, it checks the *SYS* schema for the appropriate stored prepared statement. If the statement exists, it retrieves the statement and executes it. If the statement has not yet been created and stored, the driver creates the statement, stores it in the *SYS* schema, and then uses it.

- *onDatabaseCreation*
  The JDBC driver creates all the metadata statements when Cloudscape creates a new database.

Default

Dynamic.

Example

```
cloudscape.jdbc.metadataStoredPreparedStatements=
    onDatabaseCreation
```

Scope

system-wide

NOTE: Without row locking, concurrency may be bad using the dynamic configuration.

Dynamic or Static

This property is dynamic; if you change it while Cloudscape is running, the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.language.bulkFetchDefault

Function

Sets the default number of rows that Cloudscape fetches at a time when reading a conglomerate (table or index). When this property is set to 1, Cloudscape fetches rows one at a time. Setting this property to a greater number allows Cloudscape to fetch more than one row at a time and reduces system overhead.

The optimal setting is the number of rows per page. The default value of 16 is sufficient in most situations.

To override the default value for a specific query, use the optimizer override property *bulkFetch*. See "*bulkFetch*" on page 6-3.

Bulk fetch uses memory. Cloudscape constructs a temporary cache for storing fetched rows of a size specified in the property. This cache uses memory based on the sum of the columns referenced by the scan, not the size of the entire row in the target conglomerate. Cloudscape fills the cache with the number of qualifying rows, up to the number specified with the property. Cloudscape applies simple predicates before fetching. When the scan completes, the row cache is freed.

Default

16.

Minimum Value

1.

Example

```
cloudscape.language.bulkFetchDefault=32
```

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.bulkFetchDefault', '32')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.language.bulkFetchDefault=
    '32'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

This property is dynamic; if you change it while Cloudscape is running, the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.language.defaultIsolationLevel

Function

Sets the default isolation level for the system.

READ_COMMITTED corresponds to *java.sql.Connection.TRANSACTION_READ_COMMITTED* and to ANSI level 1 and is the default isolation level for a Cloudscape system. SERIALIZABLE corresponds to *java.sql.Connection.TRANSACTION_SERIALIZABLE* (ANSI level 3).

The values are case-insensitive.

Syntax

```
cloudscape.language.defaultIsolationLevel =
{
    SERIALIZABLE |
    READ_COMMITTED
}
```

Default

READ_COMMITTED.

Example

```
-- system-wide property
cloudscape.language.defaultIsolationLevel=SERIALIZABLE

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.defaultIsolationLevel',
    'SERIALIZABLE')

-- publishing a database-wide property
ALTER PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.language.defaultIsolationLevel=
    'SERIALIZABLE'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.language.logStatementText

Function

When this property is set to true, Cloudscape writes the text and parameter values of all executed statements to the information log at the beginning of execution. It also writes information about commits and rollbacks. Information includes the time and thread number.

This property is useful for debugging.

Example

```
cloudscape.language.logStatementText=true
```

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.logStatementText', 'true')
```

```
-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.language.logStatementText=
    'true'
```

Default

False.

Scope

system-wide

database-wide (publishable)

Dynamic or Static

static

cloudscape.language.preloadClasses

Function

Specifies whether Cloudscape should synchronously compile and execute a system-configured query in order to preload compilation classes at database bootup. Setting this property to true means that boot time is slower, but the compilation time of the first query executed by the user is much faster.

A database typically boots when you first connect to it. Databases can also boot at startup; see “*cloudscape.system.bootAll*” on page 5-72.

Example

```
cloudscape.language.preloadClasses=true
```

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.preloadClasses', 'true')
```

```
-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.language.preloadClasses=
    'true'
```

Default

False.

Scope

system-wide

database-wide (publishable)

Dynamic or Static

This property is dynamic; if you change it while Cloudscape is running, the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.language.spsCacheSize

Function

Specifies the size of a database's stored prepared statement cache.

Set the cache size to the number of stored prepared statements that you regularly use, assuming that you have enough memory. If a database has fewer than 32 stored prepared statements, it is not beneficial to increase the size of this cache.

Increasing the size of a database's stored prepared statement cache is useful primarily for multi-user environments. It is also useful for single-user environments when a user wants to avoid the performance hit of preparing EXECUTE STATEMENT statements.

NOTE: This cache is flushed when you do DDL, so leave the cache at its default size when you are doing DDL.

For more information about the stored prepared statement cache, see "Additional Benefits for Multiple Connections: The Stored Prepared Statement Cache" on page 3-15.

Default

32.

Example

```
-- system-wide property
cloudscape.language.spsCacheSize=100

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.spsCacheSize',
    '100')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.language.spsCacheSize=
    '100'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

This property is static; you must reboot for changes to take effect. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.language.stalePlanCheckInterval

Function

Determines how many times Cloudscape should execute a statement before checking whether its plan is stale. When a statement is stale, Cloudscape invalidates the statement, which causes automatic recompilation the next time the statement is executed. Automatic invalidation is useful in situations when the amount of data in a table changes considerably during a single session of Cloudscape.

To read about when a statement is considered stale, see "When a Change in Table Makes a Plan Stale" on page 3-7.

For example, for a SELECT statement against an empty table, Cloudscape chooses to do a table scan instead of using an applicable index. However, as the table grows, at some point it becomes more efficient for Cloudscape to use an index to access the data. If the statement remains open, or if it is retrieved out of the statement cache or it is a stored prepared statement, however, Cloudscape continues to use the same query plan unless the statement is recompiled.

Setting the value of this property to 5 means that Cloudscape checks whether to invalidate a statement's plan after each 5 executions of the statement; setting the value of this property to *6* means that Cloudscape checks whether invalidates a statement's plan after every six executions of the statement, and so on.

To turn off automatic invalidation of statement plans, set this property to the maximum integer value (2147483647).

For more explanation, see "Understand When Statements Go Stale" on page 3-4.

NEW: The *cloudscape.language.stalePlanCheckInterval* property is new in Version 3.0.

Default

100.

Minimum Value

5

Maximum Value

java.lang.Integer.MAX_VALUE

Example

```
-- system -wide property
cloudscape.language.stalePlanCheckInterval=40

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.stalePlanCheckInterval',
    '40')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.language.stalePlanCheckInterval=
    '40'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.language.statementCacheSize

Function

Specifies the number of statements to cache per connection, which is useful if a connection recompiles *exactly the same statement* more than once. During caching, a connection attempts to reuse compiled statements rather than compile every statement anew. If Cloudscape finds an exact match for the statement, it does not have to recompile the statement.

Cached statements are aged out when the cache size is exceeded. Setting this property to zero disables caching.

Statement caching performs exact matches against incoming statements in the current connection. Statements that are not compilable are still cached.

NOTE: In situations in which you compile a prepared statement once and execute it many times, this property does not help performance. This property helps performance only in situations in which an application *compiles* exactly the same statement more than once (either executing the same *Statement* more than once, or preparing the same *PreparedStatement* more than once).

Default

20 statements.

Minimum Value

0.

Example

```
-- system-wide property
cloudscape.language.statementCacheSize=100

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.statementCacheSize',
    '100')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.language.statementCacheSize=
    '100'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

This property is static; you must reboot for changes to take effect. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.language.triggerMaximumRecursionLevel

Function

Specifies the maximum recursion level for trigger firing.

It is possible for one trigger to cause another trigger to fire, and thus it is possible for triggers to recurse infinitely. If the trigger recursion level exceeds the maximum recursion level, an exception is raised and the statement that caused the trigger to fire is rolled back.

A value of -1 means that there are no limits to recursion.

A value of 0 means that no triggers will ever fire.

NOTE: When the maximum recursion level is reached when a trigger is fired, Cloudscape throws an *SQLException* of *SQLState* X0Y73.

NEW: The *cloudscape.language.triggerMaximumRecursionLevel* property is new in Version 3.0.

Syntax

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.triggerMaximumRecursionLevel',
    'integerValue')
```

Default

16.

Minimum Value

-1 (no limit; see notes above).

Example

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.language.triggerMaximumRecursionLevel',
    '10')

-- publishing a database-wide property
CREATE PUBLICATION APub
```

```
ADD TARGET DATABASE PROPERTY
cloudscape.language.triggerMaximumRecursionLevel=
    '10'
```

Scope

database-wide (publishable)

system-wide

Dynamic or Static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.locks.deadlockTimeout

Function

Determines the number of seconds after which Cloudscape checks whether a transaction waiting to obtain a lock is involved in a deadlock. If a deadlock has occurred, and Cloudscape chooses the transaction as a deadlock victim, Cloudscape aborts the transaction. The transaction receives an *SQLException* of *SQLState* 40001. If the transaction is not chosen as the victim, it continues to wait for a lock if *cloudscape.locks.waitTimeout* is set to a higher value than the value of *cloudscape.locks.deadlockTimeout*.

If this property is set to a higher value than *cloudscape.locks.waitTimeout*, no deadlock checking occurs. See "*cloudscape.locks.waitTimeout*" on page 5-51.

NEW: The behavior of the *cloudscape.locks.deadlockTimeout* property is new in Version 3.0. In prior versions, this property specified the amount of time a transaction waited for a lock before Cloudscape assumed that a deadlock occurred. In Version 3.0, Cloudscape provides real deadlock checking.

For more information about deadlock checking, see "Deadlocks" on page 6-22

Default

60 seconds.

Example

```
cloudscape.locks.deadlockTimeout=30
```

```
-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.locks.deadlockTimeout', '30')
```

```
-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.locks.deadlockTimeout=
    '30'
```

Scope

database-wide (publishable)

system-wide

Dynamic or Static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.locks.deadlockTrace

Function

Causes a stack trace of all threads involved in deadlocks (not just the victims) to be written to the error log (typically the *cloudscape.LOG* file). This property is meaningful only if the *cloudscape.locks.monitor* property is set to *true*.

NOTE: This level of debugging is intrusive: it may alter the timing of the application, reduce performance severely, and produce a large error log file. It should be used with care.

NEW: The *cloudscape.locks.deadlockTrace* property is new in Version 3.0.

Default

False.

Example

```
-- system property
cloudscape.locks.deadlockTrace=true

CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.locks.deadlockTrace', 'true')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.locks.deadlockTrace=
    'true'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.locks.escalationThreshold

Function

Sets the threshold at which the optimizer (during compilation) escalates the locking on a table from row level to table level for a statement. If the optimizer estimates that executing the statement will lock more rows in a table than the number specified in this property, the optimizer locks the entire table instead and releases the row-level locks.

Also used by the Cloudscape system at runtime in determining when to attempt to escalate locking for at least one of the tables involved in a transaction from row-level locking to table-level locking. This runtime decision is independent of any compilation decision.

A large number of row locks use a lot of resources. If nearly all the rows are locked, it may be worth the slight decrease in concurrency to lock the entire table to avoid the large number of row locks.

For more information, see “Locking and Performance” on page 4-28.

It is useful to increase this value for large systems (such as enterprise-level servers, where there is more than 64 MB of memory), and to decrease it for very small systems (such as palmtops).

Syntax

```
cloudscape.locks.escalationThreshold=numberOfLocks
```

Default

5000.

Minimum

100.

Maximum

2,147,483,647.

Example

```
-- system-wide property
cloudscape.locks.escalationThreshold=1000

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.locks.escalationThreshold',
    '1000')

-- publishing a database-wide property
ALTER PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.locks.escalationThreshold=
    '1000'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.locks.monitor

Function

Specifies that all deadlock errors are logged to the error log. If *cloudscape.stream.error.logSeverityLevel* is set to ignore deadlock errors, *cloudscape.locks.monitor* overrides it.

NEW: The *cloudscape.locks.monitor* property is new in Version 3.0.

Default

False.

Example

```
-- system property
cloudscape.locks.monitor=true

CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.locks.monitor', 'true')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.locks.monitor=
    'true'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.locks.waitTimeout

Function

Specifies the number of seconds after which Cloudscape aborts a transaction when it is waiting for a lock. When Cloudscape aborts (and rolls back) the transaction, the transaction receives an *SQLException* of *SQLState* 40XL1.

The time specified by this property is approximate.

A zero value for this property means that Cloudscape aborts a transaction any time it cannot immediately obtain a lock that it requests.

A negative value for this property is equivalent to an infinite wait time; the transaction waits forever to obtain the lock.

If this property is set to a value greater than or equal to zero but less than the value of *cloudscape.locks.deadlockTimeout*, then Cloudscape never performs any deadlock checking.

NEW: The *cloudscape.locks.waitTimeout* property is new in Version 3.0.

Default

180 seconds.

Example

```
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.locks.waitTimeout', '15')
cloudscape.locks.waitTimeout=60

-- publishing a database-wide property
ALTER PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.locks.waitTimeout=
    '15'
```

Scope

database-wide (publishable)

system-wide

Dynamic or Static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.service

Function

The *cloudscape.service* property specifies that a database stored in a directory other than the system directory, or stored in a subdirectory of the system directory, is part of the current Cloudscape system when it starts. (Databases that are directly in the system directory are always part of the current Cloudscape system when it starts.)

(A database that is part of the current system directory, including those specified by this property, does not automatically boot at startup unless you use the *cloudscape.system.bootAll* property.)

For this property, specify the relative or absolute path to the database's directory, known as a *serviceDirectory.* Use the same conventions as for connecting to a database outside the system; see "Conventions for Specifying the Database Path" on page 2-18 in the *Cloudscape Developer's Guide.* For example, for a database called *newDB* in the directory *trial_directory* on the C drive, its *serviceDirectory,* specified as an absolute path, would be *c:/trial_directory/newDB.* You can also specify a relative path.

If you do not set this property, such a database is considered part of the system only when a connection is first made to it.

The database must exist when Cloudscape starts.

NOTE: When you set this property in the *cloudscape.properties* file, avoid using absolute paths, because Java interprets colons as being equivalent to an equal sign (the separator between the property name and its value) when properties are set in properties files. This means that colons are interpreted incorrectly. The problem does not occur when you are setting properties on the command line or in a JDK 1.2 environment.

Syntax

```
cloudscape.service.databaseDirectory=serviceDirectory
```

Examples

```
cloudscape.service.london/sales=serviceDirectory
cloudscape.service.demo/databases/toursDB=serviceDirectory
```

Scope

system-wide

Dynamic or Static

This property is dynamic; if you change it while Cloudscape is running, the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.storage.initialPages

Function

The on-disk size of a Cloudscape table grows by one page at a time until eight pages of user data (or nine pages of total disk use, one is used for overhead) have been allocated. Then it will grow by eight pages at a time if possible.

A Cloudscape table or index can be created with a number of pages already pre-allocated. To do so, specify the property in the CREATE TABLE or CREATE INDEX PROPERTIES list.

Define the number of user pages the table or index is to be created with. The purpose of this property is to preallocate a table or index of reasonable size if the user expects that a large amount of data will be inserted into the table or index. A table or index that has the preallocated pages will enjoy a small performance improvement over a table or index that has no preallocated pages when the data are loaded.

The total desired size of the table or index should be

```
(1+cloudscape.storage.initialPages) *
cloudscape.storage.pageSize bytes.
```

When you create a table or an index using this property, Cloudscape attempts to allocate the requested number of user pages. However, the operations do not fail even if they are unable to allocate the requested number of pages, as long as they allocate at least one page.

Default

1 page.

Minimum Value

The minimum number of *initialPages* is 1.

Maximum Value

The maximum number of *initialPages* is 1000.

Example

```
cloudscape.storage.initialPages=30
```

```
CREATE TABLE newTable (i INT, j VARCHAR(50))
PROPERTIES cloudscape.storage.initialPages = 30
```

Scope

conglomerate-specific

cloudscape.storage.minimumRecordSize

Indicates the minimum user row size in bytes for on-disk database pages for tables when you are creating a table. This property ensures that there is enough room for a row to grow on a page when updated without having to overflow. This is generally most useful for VARCHAR and BIT VARYING data types and for tables that are updated a lot, in which the rows start small and grow due to updates. Reserving the space at the time of insertion minimizes row overflow due to updates, but it can result in wasted space.

See also "*cloudscape.storage.pageReservedSpace*" on page 5-60.

Valid Conglomerates

Tables only.

Default

12 bytes.

Minimum

12 bytes.

Maximum

cloudscape.storage.pageSize * (1 - *cloudscape.storage.pageReservedSpace*/100) – 100.

If you set this property to a value outside the legal range, Cloudscape uses the default value.

Examples

```
-- changing the default for the system
cloudscape.storage.minimumRecordSize=128

-- changing the default for the database
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.storage.minimumRecordSize',
    '128')

-- changing the default for target databases
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
```

```
cloudscape.storage.minimumRecordSize=
    '128'

-- overriding the defaults for a single table
CREATE TABLE A(textcol LONG VARCHAR)
PROPERTIES cloudscape.storage.minimumRecordSize=128
```

Scope

system-wide

database-wide (publishable)

conglomerate-specific

Dynamic or Static

This property is dynamic; if you change it while Cloudscape is running, the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.storage.pageCacheSize

Function

Defines the size, in number of pages, of the database's data page cache (data pages kept in memory).

The actual amount of memory the page cache will use depends on the following:

- the size of the cache (configured with this property, *cloudscape.storage.pageCacheSize*)
- the size of the pages (configured with the *cloudscape.storage.pageSize* property)
- overhead (varies with JVMs)

When increasing the size of the page cache, you typically have to allow more memory for the Java heap when starting the embedding application (taking into consideration, of course, the memory needs of the embedding application as well). For example, using the default page size of 4K, a page cache size of 2000 pages will require at least 8 MB of memory (and probably more, given the overhead).

For a simple application (no GUI), using the Javasoft 1.1.7 JVM on Windows NT and using the -mx96m option (which allows 96 MB for the Java heap), it is possible to have a page cache size of 10,000 pages (approximately 40 MB).

Default

40 pages.

Example

```
cloudscape.storage.pageCacheSize=160
```

Scope

system-wide

Dynamic or Static

Static. You must reboot the system for the change to take effect.

cloudscape.storage.pageReservedSpace

Function

Defines the percentage of space reserved for updates on an on-disk database page for tables only (not indexes); indicates the percentage of space to keep free on a page when inserting. Leaving reserved space on a page can minimize row overflow (and the associated performance hit) during updates. Once a page has been filled up to the reserved-space threshold, no new rows are allowed on the page. This reserved space is used only for rows that increase in size when updated, not for new inserts.

Regardless of the value of *cloudscape.storage.pageReservedSpace*, an empty page always accepts at least one row.

Valid Conglomerates

Tables only.

Default

20%.

Minimum Value

The minimum value is 0% and the maximum is 100%. If you specify a value outside this range, Cloudscape uses the default value of 20%.

Example

```
-- modifying the default for the system
cloudscape.storage.pageReservedSpace=40

-- modifying the default for the database
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.storage.pageReservedSpace',
    '40')

-- overriding the defaults for a single table
CREATE TABLE B(textcol LONG VARCHAR)
PROPERTIES cloudscape.storage.pageReservedSpace=40
```

Scope

system-wide

database-wide

conglomerate-specific

Dynamic or Static

This property is dynamic; if you change it while Cloudscape is running, the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.storage.pageSize

Function

Defines the page size, in bytes, for on-disk database pages for tables or indexes used during table or index creation. Page size should be a power of 2.

Valid Conglomerates

Tables and indexes, including the indexes created to enforce constraints.

Default

4096 bytes.

Minimum Value

The minimum size is 1024 bytes. If you specify a value lower than the minimum, Cloudscape uses the default value 4096.

Maximum Value

None.

Example

```
-- changing the default for the system
cloudscape.storage.pageSize=8192

-- changing the default for the database
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.storage.pageSize',
    '8192')

-- changing the default for target databases
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY cloudscape.storage.pageSize=
    '8192'

-- overriding the defaults for a single table
CREATE TABLE A(textcol LONG VARCHAR)
PROPERTIES cloudscape.storage.pageSize=8192
```

Scope

system-wide

database-wide (publishable)

conglomerate-specific

Dynamic or Static

This property is dynamic; if you change it while Cloudscape is running, the change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

cloudscape.storage.rowLocking

Function

Use this property to disable row-level locking. Row-level locking uses more system resources, such as disk space and memory, so if an application does not need row-level locking, you can use this system property to force all locking to table level. For example, read-only and single-user applications probably do not need row-level locking.

This property can be set to one of the boolean values *true* and *false*. Setting the property to true enables row-level locking (the default behavior). Setting the property to false disables row-level locking.

Default

True.

Example

```
cloudscape.storage.rowLocking=false
```

```
-- database-level property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.storage.rowLocking',
    'false')
```

```
-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.storage.rowLocking=
    'false'
```

Scope

system-wide

database-wide (publishable)

NEW: The ability to set the *cloudscape.storage.rowLocking* property as a database-wide property is new in Version 3.0.

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.storage.tempDirectory

Function

Defines the location on disk for temporary file space needed by Cloudscape for performing large sorts and deferred deletes and updates. (Temporary files are automatically deleted after use, and are removed when the database restarts after a crash.) The temporary directory named by this property will be created if it does not exist, but will not be deleted when the system shuts down. The path name specified by this property must have file separators that are appropriate to the current operating system.

This property allows databases located on read-only media to write temporary files to a writable location. If this property is not set, databases located on read-only media may get an error like the following:

```
ERROR XSDF1: Exception during creation of file
c:\databases\db\tmp\T887256591756.tmp for container
ERROR XJ001: Java exception:
'a:\databases\db\tmp\T887256591756.tmp: java.io.IOException'.
```

This property moves the temporary directories for all databases being used by the Cloudscape system. Cloudscape makes temporary directories for each database under the directory referenced by this property. For example, if the property is set as follows:

```
cloudscape.storage.tempDirectory=C:/Temp/dbtemp
```

the temporary directories for the databases in *C:\databases\db1* and *C:\databases\db2* will be in *C:\Temp\dbtemp\db1* and *C:\Temp\dbtemp\db2*, respectively.

The temporary files of two databases running concurrently with the same name (e.g., *C:\databases\db1* and *E:\databases\db1*) will conflict with each other if the *cloudscape.storage.tempDirectory* property is set. This will cause incorrect results, so users are advised to give databases unique names.

Default

A subdirectory named *tmp* under the database directory.

For example, if the database *db1* is stored in *C:\databases\db1*, the temporary files are created in *C:\databases\db1\tmp*.

Example

```
-- system-wide property
cloudscape.storage.tempDirectory=c:/Temp/dbtemp

-- database-wide property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.storage.tempDirectory',
    'c:/Temp/dbtemp')
```

Scope

system-wide

database-wide (not publishable)

Dynamic or Static

This property is static; you must restart Cloudscape for a change to take effect.

cloudscape.stream.error.field

Function

Specifies a static field that references a stream to which the error log is written. The field is specified using the fully qualified name of the class, then a dot (.) and then the field name. The field must be public and static. Its type can be either *java.io.OutputStream* or *java.io.Writer.*

The field is accessed once at Cloudscape boot time, and the value is used until Cloudscape is rebooted. If the field returns, the error stream defaults to the system error stream (*java.lang.System.err*).

If the field does not exist or is inaccessible, the error stream defaults to the system error stream. Cloudscape will not call the *close()* method of the object obtained from the field.

Default

None.

Example

```
cloudscape.stream.error.field=java.lang.System.err
```

Scope

system-wide

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.stream.error.file

Function

Specifies name of the file to which the error log is written. If the file name is relative, it is taken as relative to the system directory.

If this property is set, the properties *cloudscape.stream.error.method* and *cloudscape.stream.error.field* are ignored.

Default

cloudscape.LOG.

Example

```
cloudscape.stream.error.file=error.txt
```

Scope

system-wide

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.stream.error.logSeverityLevel

Function

Specifies which errors are logged to the Cloudscape error log (typically the *cloudscape.LOG* file).

Any error raised in a Cloudscape system is given a level of severity. This property indicates the minimum severity necessary for an error to appear in the error log. The severities are defined in the class *COM.cloudscape.types.JBMSExceptionSeverity.* The higher the number, the more severe the error.

- *20000*
  Errors that cause the statement to be rolled back, for example syntax errors and constraint violations.
- *30000*
  Errors that cause the transaction to be rolled back, for example deadlocks.
- *40000*
  Errors that cause the connection to be closed.
- *50000*
  Errors that shut down the Cloudscape system.

Default

40000.

Example

```
// send errors of level 30000 and higher to the log
cloudscape.stream.error.logSeverityLevel=30000
```

Scope

system-wide

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.stream.error.method

Function

Specifies a static method that returns a stream to which the Cloudscape error log is written.

Specify the method using the fully qualified name of the class, then a dot (.) and then the method name. The method must be public and static. Its return type can be either *java.io.OutputStream* or *java.io.Writer.* Cloudscape will not call the *close()* method of the object returned by the method.

The method is called once at Cloudscape boot time, and the return value is used for the lifetime of Cloudscape. If the method returns null, the error stream defaults to the system error stream. If the method does not exist or is inaccessible, the error stream defaults to the system error stream (*java.lang.System.err*).

If the value of this property is set, the property *cloudscape.stream.error.field* is ignored.

Default

Not set.

Example

```
cloudscape.stream.error.method=java.sql.DriverManager.
getLogStream
```

Scope

system-wide

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.system.bootAll

Function

Specifies whether the databases that are part of the current system are automatically booted when the Cloudscape system starts. When this property is set to false, a database is booted only when a connection is first made.

Databases that are not part of the current system are booted only upon the first connection.

NOTE: For information on what databases are considered part of the current system when it starts up, see "*cloudscape.service*" on page 5-53.

Cloudscape recommends that you set this property to *true* when you use Cloudscape as a database server to multiple clients and *false* when you use Cloudscape embedded in a single-user application.

The Cloudscape *system* boots when the local JDBC driver is loaded. Booting databases is a separate step in which Cloudscape checks whether recovery needs to be run on the databases. For information about recovery, see the *Cloudscape Developer's Guide*.

Encrypted databases cannot be booted with this property. Encrypted databases must be booted individually by connecting to them with a valid key.

Default

False.

Example

```
cloudscape.system.bootAll=true
```

Scope

system-wide

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.system.home

Function

Specifies the Cloudscape system directory, which is the directory that contains subdirectories holding databases that you create and the text file *cloudscape.properties*.

If the system directory that you specify with *cloudscape.system.home* does not exist at startup, Cloudscape creates the directory automatically.

Default

Current directory (the value of the JVM system property *user.dir*).

If you do not explicitly set the *cloudscape.system.home* property when starting Cloudscape, the default is the directory in which Cloudscape was started.

NOTE: Cloudscape recommends that you always explicitly set the value of *cloudscape.system.home*.

Example

```
-Dcloudscape.system.home=C:\cloudscape
```

Scope

system-wide, Set programmatically only

Dynamic or Static

This property is static; if you change it while Cloudscape is running, the change does not take effect until you reboot.

cloudscape.user.*UserName*

Function

Has two uses:

- Creates users and passwords when *cloudscape.authentication.provider* is set to *CLOUDSCAPE*.
- Caches user DNs locally when *cloudscape.authentication.provider* is set to *LDAP* and *cloudscape.authentication.ldap.searchFilter* is set to *cloudscape.user*.

Users and Passwords

This property creates valid clear-text users and passwords within Cloudscape when the *cloudscape.authentication.provider* property is set to *CLOUDSCAPE*. For information about users, see “Working with User Authentication” on page 8-5 of the *Cloudscape Developer’s Guide*.

- *Database-Level Properties*
  When you create users with database-level properties, those users are available to the specified database only.

  You set the property once for each user. To delete a user, set that user’s password to null.
- *System-Level Properties*
  When you create users with system-level users, those users are available to all databases in the system.

  You set the value of this system-wide property once for each user, so you may set it several times. To delete a user, remove that user from the file.

  You can define this property in the usual ways—typically in the *cloudscape.properties* file.

When a user name and its corresponding password are provided in the *DriverManager.getConnection* call, Cloudscape validates them against the properties defined for the current system.

User names are *SQL92Identifiers* and can be delimited.

Caching User DNs

This property caches user DNs (distinguished names) locally when *cloudscape.authentication.provider* is set to *LDAP* and

cloudscape.authentication.ldap.searchFilter is set to *cloudscape.user.* When you provide a user DN with this property, Cloudscape is able to avoid an LDAP search for that user's DN before authenticating. For those users without DNs defined with this property, Cloudscape performs a search using the default value of *cloudscape.authentication.ldap.searchFilter*.

Syntax

```
cloudscape.user.{UserName=Password} | UserName=userDN }
```

```
-- database-level property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.user.UserName',
    'Password | userDN')
```

Default

None.

Examples

```
-- system-level property
cloudscape.user.guest=java5w
cloudscape.user.sa=cloud3x9
cloudscape.user."!Amber"=java5w

-- database-level property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.user.sa',
    'cloud3x9')

-- cache a userDN locally
cloudscape.user.richard=uid=richard, ou=People,
o=cloudscape.com

-- cache a userDN locally, database-level property
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.user.richard',
    'uid=richard, ou=People, o=cloudscape.com')

-- publishing a database-wide property
CREATE PUBLICATION APub
ADD TARGET DATABASE PROPERTY
cloudscape.user.guest=
    'guestPW'
```

Scope

system-wide

database-wide (publishable)

Dynamic or Static

Dynamic. The change takes effect immediately. For information about dynamic changes to properties, see “Dynamic or Static Changes to Properties” on page 1-10.

6 Optimizer Overrides

Cloudscape's query optimizer usually makes the best choice of join order and access path. In addition, the default values for join strategy and row fetch are usually best. However, there are some cases in which you may want to override the optimizer or the default values. Optimizer overrides allow users to hand-tune the optimizer for queries and updates and deletes with WHERE clauses.

Optimizer overrides are specified within a PROPERTIES clause within the SQL-J statement.

A PROPERTIES clause has the following syntax:

```
PROPERTIES propertyName = value [, propertyName = value]*
```

A *propertyName* is case-sensitive.

An exception will be thrown if there are semantic errors in the PROPERTIES clause such as:

- invalid properties
- nonexistent property values, such as nonexistent access paths
- invalid property values, such as invalid access paths

Optimizer-override properties belong to one of the following scopes:

- an entire FROM clause
- a particular table in the FROM clause

FROM clause optimizer-override PROPERTIES clauses, which belong to an entire FROM clause, come immediately after the word FROM, like this:

```
FROM [ PROPERTIES joinOrder = { FIXED | UNFIXED } ]
    TableExpression [,TableExpression]*
```

Table optimizer-override PROPERTIES clauses, which belong to a particular table in the FROM clause, come at the end of the *TableExpression*, like this:

```
{ TableName | ViewName  }
    [ [ AS ] CorrelationName
        [ (SimpleColumnName [ , SimpleColumnName]* ) ] ] ]
[ PROPERTIES clause ]
```

For more information about a *TableExpression*, see “TableExpression” on page 1-116 in the *Cloudscape Reference Manual*.

With the exception of *joinStrategy* (which is allowed on all table expressions), such properties can be specified only on base tables; they are not allowed on views or derived tables. They are allowed on any base table within a FROM clause or any table within a FROM clause in a subquery. They can be used on the base tables specified in a CREATE VIEW statement, within a derived table (subquery in the FROM list), and within a JOIN clause.

Errors for table optimizer override PROPERTIES clauses in a CREATE VIEW statement are not returned until the view is first used in a DML statement.

Optimizer-override properties include:

- “*bulkFetch*” on page 6-3
- “*constraint*” on page 6-5
- “*index*” on page 6-8
- “*joinOrder*” on page 6-9
- “*joinStrategy*” on page 6-10

bulkFetch

Function

The *bulkFetch* optimizer property overrides the default number of rows that Cloudscape fetches at a time when reading a conglomerate (table or index). When this property is set to 1, Cloudscape fetches rows one at a time. Setting this property to a greater number allows Cloudscape to fetch more than one row at a time and reduces system overhead.

Setting the *bulkFetch* property uses memory. Cloudscape constructs a temporary cache for storing fetched rows of a size specified in the property. This cache uses memory based on the sum of the columns referenced by the scan, not the size of the entire row in the target conglomerate. Cloudscape fills the cache with the number of qualifying rows, up to the number specified with the property. Cloudscape applies simple predicates (such as WHERE Region = 'Europe') before fetching. When the scan completes, the row cache is freed.

Setting the *bulkFetch* property to a large value (up to the number of rows in a typical page) enhances performance at the expense of the amount of memory needed for the row cache. There is no performance gain in setting the *bulkFetch* size larger than the number of qualifying rows from the target table. For example, if the query will return only 5 rows, there is no performance benefit in setting this property to 30.

The default bulk fetch size is determined by the *cloudscape.language.bulkFetchDefault* system property; the default value for that property is 16. That value is optimal for most situations. For more information, see "*cloudscape.language.bulkFetchDefault*" on page 5-31.

Syntax

```
bulkFetch=size
```

Default Value

16.

Minimum Value

1.

Maximum Value

java.lang.Integer.MAX_INTEGER.

Example

```
SELECT *
FROM FlightAvailability
PROPERTIES bulkFetch=256
```

Restrictions

Bulk fetch is not used in the following cases:

- with a *joinStrategy=hash* clause, since the hash join strategy always causes the entire table to be fetched at once
- with updatable cursors, since a positioned update or delete requires the current scan position to locate a single target row
- with nonmaterialized subqueries
- if the table in question is the inner table of an equijoin on a unique key

When Useful

Bulk fetch is best suited for situations in which many rows are returned.

For example, it is useful for table scans or index range scans:

```
SELECT *
FROM Flights
WHERE orig_airport > 'AAB'
AND orig_airport < 'XYZ'
```

```
SELECT *
FROM Flights
```

```
SELECT *
FROM Flights
WHERE segment_number = 1
```

Scope

Table optimizer-override property.

constraint

Function

The Cloudscape optimizer chooses an index, including the indexes that enforce constraints, as the access path for query execution if it is useful. If there is more than one useful index, in most cases Cloudscape chooses the index that is most useful.

You can override the optimizer's selection and force use of a particular index or force a table scan. To force use of the index that enforces a primary key or unique constraint, use this property, specifying the unqualified name of the constraint.

System-generated constraint and index names use lowercase letters, so you must treat them as delimited identifiers and enclose them in double quotation marks.

NOTE: An exception is thrown if the access path is nonexistent or invalid. For example, the optimizer cannot use an index if the statement is an updatable cursor whose updatable columns are in the index's key, or if the statement is an UPDATE statement and the updated columns are in the index's key.

Syntax

constraint=*constraintName*

Example

```
-- specifying the index or constraint affects the access
-- path during query execution
CREATE TABLE mytable (a int, b int, c int,
CONSTRAINT mykey primary key (a, b));
0 rows inserted/updated/deleted
ij> INSERT INTO mytable values (1, 2, 3), (1, 3, 3), (2, 2, 3);
3 rows inserted/updated/deleted
ij> CREATE BTREE INDEX myindex ON mytable(a);
0 rows inserted/updated/deleted
ij>  -- there are now two useful indexes on the
-- values in column a
SET RUNTIMESTATISTICS ON;
0 rows inserted/updated/deleted
ij> -- let the optimizer choose an access path
-- it will use one of the two useful indexes
SELECT * FROM mytable WHERE a = 1;
```

```
A              |B              |C
-----------------------------------------------
1              |2              |3
1              |3              |3

2 rows selected
ij>  -- runtimestatistics() will show the access path
VALUES RUNTIMESTATISTICS().toString();
. . .
IndexRowToBaseRowResultSet for MYTABLE:
. . .

        IndexScanResultSet for MYTABLE using constraint MYKEY . . .

1 row selected
ij>  -- force a table scan (use no index)
SELECT * FROM mytable
PROPERTIES index=NULL
WHERE a = 1;
A              |B              |C
-----------------------------------------------
1              |2              |3
1              |3              |3

2 rows selected
ij> VALUES RUNTIMESTATISTICS().toString();
. . .
TableScanResultSet for MYTABLE . . .
. . .
1 row selected
ij>
-- force the use of myindex
SELECT * FROM mytable
PROPERTIES index=myindex
WHERE a = 1;
A              |B              |C
-----------------------------------------------
1              |2              |3
1              |3              |3
```

```
2 rows selected
ij> VALUES RUNTIMESTATISTICS().toString();
. . .
IndexRowToBaseRowResultSet for MYTABLE:
. . .
        IndexScanResultSet for MYTABLE using index MYINDEX . . .

1 row selected
ij> -- force the use of the (named) primary key constraint

SELECT * FROM mytable
PROPERTIES constraint=mykey
WHERE a = 1;
A              |B              |C
-----------------------------------------------
1              |2              |3
1              |3              |3

2 rows selected
ij> VALUES RUNTIMESTATISTICS().toString();
. . .
IndexRowToBaseRowResultSet for MYTABLE:
        IndexScanResultSet for MYTABLE using constraint MYKEY . . .
               next time in milliseconds/row = 0

1 row selected
ij>
```

Scope

Table optimizer-override property.

index

Function

The Cloudscape optimizer chooses an index, including the indexes that enforce constraints, as the access path for query execution if it is useful. If there is more than one useful index, in most cases Cloudscape chooses the index that is most useful.

You can override the optimizer's selection and force use of a particular index or force a table scan. To force use of a particular index, specify the unqualified index name. To force a table scan, specify null for the index name.

See also "*constraint*" on page 6-5.

System-generated index names use lowercase letters, so you must treat them as delimited identifiers and enclose them in double quotation marks.

NOTE: An exception is thrown if the access path is nonexistent or invalid. The optimizer cannot use an index if the statement is an updatable cursor whose updatable columns are in the index's key, or if the statement is an UPDATE statement and the updated columns are in the index's key.

Syntax

```
index= { indexName | null }
```

Example

See the example under "*constraint*" on page 6-5.

Scope

Table optimizer-override property.

joinOrder

Function

Allows you to override the optimizer's choice of join order for two tables. When you use the value FIXED, the optimizer chooses the order of tables as they appear in the FROM clause as the join order.

Syntax

```
joinOrder = { FIXED | UNFIXED }
```

FIXED means the optimizer should use the order of items as they appear in the FROM clause in the statement; UNFIXED (which is the default) allows the optimizer to make its own choice about join order.

The words *FIXED* and *UNFIXED* are case-insensitive.

Default

UNFIXED.

Example

```
SELECT *
FROM PROPERTIES joinOrder = FIXED
    Flights AS fts, FlightAvailability AS fa
WHERE fts.flight_id = fa.flight_id
AND fts.segment_number = fa.segment_number
```

Scope

FROM clause optimizer-override property.

joinStrategy

Function

The *joinStrategy* property allows you to override the optimizer's choice of join strategy. The two types of join strategy are called *nested loop* and *hash*. In a nested loop join strategy, for each qualifying row in the outer table, Cloudscape uses the appropriate access path (index or table scan) to find the matching rows in the inner table. In a hash join strategy, Cloudscape constructs a hash table representing the inner table. For each qualifying row in the outer table, Cloudscape does a quick lookup on the hash table to find the matching rows in the inner table. Cloudscape has to scan the inner table or index only once to create the hash table.

Nested loop joins are useful in most situations.

Hash joins are useful in situations in which the inner table values are unique and there are many qualifying rows from the outer table.

The PROPERTIES clause must appear directly after the *inner* table.

NEW: Beginning in Version 3.0, you are allowed to specify a hash join for a virtual table such as a derived table, view, or external virtual table.

NOTE: Use this optimizer override with the *joinOrder* property only. Do not let the optimizer choose the join order.

NOTE: When the optimizer automatically considers a hash join, it knows not to choose hash join if it estimates that the amount of memory required to build the hash table would exceed 1 MB. Forcing a hash join with this property does not take such memory use into consideration, and you can run out of memory.

Syntax

```
joinStrategy={NESTEDLOOP | HASH}
```

The words *NESTEDLOOP* and *HASH* are case-insensitive.

Default

Chosen by the optimizer.

Example

```
-- FlightAvailability is the inner table, so we specify
-- the joinStrategy after it
SELECT *
FROM PROPERTIES joinOrder = FIXED
    Flights AS fts, FlightAvailability AS fa
PROPERTIES joinStrategy=HASH
WHERE fts.flight_id = fa.flight_id
AND fts.segment_number = fa.segment_number

--segments_seatbookings is a view
SELECT * FROM PROPERTIES joinOrder=FIXED customizedtours,
segments_seatbookings
PROPERTIES joinStrategy=NESTEDLOOP WHERE
customized_tour->begin = travel_date
```

Scope

Table optimizer-override property.

Appendix A Internal Language Transformations

The Cloudscape SQL-J parser sometimes transforms SQL-J statements internally for performance reasons. This appendix describes those transformations. Understanding the internal language transformations may help you analyze and tune performance. Understanding the internal language transformations is not necessary for the general user.

- "Predicate Transformations" on page A-2
- "Transitive Closure" on page A-7
- "View Transformations" on page A-9
- "Subquery Processing and Transformations" on page A-11
- "Sort Avoidance" on page A-17
- "Aggregate Processing" on page A-21

This chapter uses some specialized terms. Here are some definitions:

base table	A real table in a FROM list. In queries that involve "virtual" tables such as views and derived tables, base tables are the underlying tables to which virtual tables correspond.
derived table	A virtual table, such as a subquery given a correlation name or a view. For example: *SELECT derivedtable.c1 FROM (VALUES 'a', 'b') AS derivedTable (c1, c2)*.
equality predicate	A predicate in which one value is compared to another value using the = operator.
equijoin predicate	A predicate in which one column is compared to a column in another table using the = operator.
optimizable	A predicate is *optimizable* if it provides a starting or stopping point and allows use of an index. Optimizable predicates use only simple column references and =, <, >, +, >=, and IS NULL operators. For complete details, see "What's Optimizable?" on page 4-5. A synonym for *optimizable* is *indexable*.
predicate	A WHERE clause contains boolean expressions that can be linked together by AND or OR clauses. Each part is called a *predicate*. For example: *WHERE c1 =2 AND c2 = 5* contains two predicates.
sargable	*Sargable* predicates are a superset of optimizable predicates; not all sargable predicates are optimizable, because sargable predicates also include the <> operator. (*Sarg* stands for "search argument.") Predicates that are sargable but not optimizable nevertheless improve performance and allow the optimizer to use more accurate costing information. In addition, sargable predicates can be *pushed down* (see "Predicates Pushed into Views or Derived Tables" on page A-10).
simple column reference	A reference to a column that is not part of an expression. For example, *c1* is a simple column reference, but *c1+1*, *max(c1)*, and *c1.toString()* are not.

Predicate Transformations

WHERE clauses with predicates joined by OR are usually not optimizable. WHERE clauses with predicates joined by AND are optimizable if *at least one* of the predicates is optimizable. For example:

```
SELECT * FROM Flights
WHERE flight_id = 'AA1111'
AND segment_number <> 2
```

In this example, the first predicate is optimizable; the second predicate is not. Therefore, the statement is optimizable.

NOTE: In a few cases, a WHERE clause with predicates joined by OR can be transformed into an optimizable statement. See "OR Transformations" on page A-6.

Cloudscape can transform some predicates internally so that at least one of the predicates is optimizable and thus the statement is optimizable. This section describes the predicate transformations that Cloudscape performs to make predicates optimizable.

A predicate that uses the following comparison operators can sometimes be transformed internally into optimizable predicates:

- "BETWEEN Transformations" on page A-3
- "LIKE Transformations" on page A-3
- "Static IN Predicate Transformations" on page A-5
- "OR Transformations" on page A-6
- "Transitive Closure" on page A-7

BETWEEN Transformations

A BETWEEN predicate is transformed into equivalent predicates that use the >= and <= operators, which are optimizable. For example:

```
booking_date BETWEEN DATE'1998-12-01' AND DATE'1998-12-15'
```

is transformed into

```
booking_date >= DATE'1998-12-01'
AND booking_date <= '1998-12-15'
```

LIKE Transformations

- "Character String Beginning with Constant" on page A-4
- "Character String Without Wildcards" on page A-4
- "Unknown Parameter" on page A-5

Character String Beginning with Constant

A LIKE predicate in which a column is compared to a character string that *begins* with a character constant (not a wildcard) is transformed into three predicates: one predicate that uses the LIKE operator, one that uses the >= operator, and one that uses the < operator. For example:

```
country LIKE 'Ch%i%'
```

becomes

```
country LIKE 'Ch%i%'
AND country >= 'Ch'
AND country < 'Ci'
```

The first (LIKE) predicate is not optimizable, but the new predicates added by the transformation are.

When the character string begins with one more character constants and ends with a single "%", the first LIKE clause is eliminated. For example:

```
country LIKE 'Ch%'
```

becomes

```
country >= 'Ch'
AND country < 'Ci'
```

Character String Without Wildcards

A LIKE predicate is transformed into a predicate that uses the = operator (and a NOT LIKE predicate is transformed into one that uses <>) when the character string does not contain any wildcards. For example:

```
country LIKE 'Chile'
```

becomes

```
country = 'Chile'
```

and

```
country NOT LIKE 'Chile'
```

becomes

```
country <> 'Chile'
```

Predicates that use the = operator are optimizable. Predicates that use the <> operator are sargable.

Unknown Parameter

The situation is similar to those described above when a column is compared using the LIKE operator to a parameter whose value is unknown in advance (dynamic parameter, join column, etc.).

In this situation, the LIKE predicate is likewise transformed into three predicates: one LIKE predicate, one predicate using the >= operator, and one predicate using the < operator. For example:

```
country LIKE ?
```

is transformed into

```
country LIKE ?
AND country >= InternallyGeneratedParameter
AND country < InternallyGeneratedParameter
```

where the *InternallyGeneratedParameters* are calculated at the beginning of execution based on the value of the parameter.

NOTE: This transformation can lead to a bad plan if the user passes in a string that begins with a wildcard or a nonselective string as the parameter. Users can work around this possibility by writing the query like this (which is not optimizable):

```
(country || '') LIKE ?
```

Static IN Predicate Transformations

A static IN list predicate is one in which the IN list is composed entirely of constants. Cloudscape calculates the minimum and maximum values in the list and transforms the predicate into three new predicates: the original IN predicate, one that uses the >= operator, and one that uses the <= operator. The second and third are optimizable. For example:

```
orig_airport IN ('ABQ', 'AKL', 'DSM')
```

is transformed into

```
orig_airport IN ('ABQ', 'AKL', 'DSM')
AND orig_airport >= 'ABQ'
AND orig_airport <= 'DSM'
```

NOT IN Predicate Transformations

NOT IN lists are transformed into multiple predicates that use the <> operator. <> predicates are not optimizable, but they are sargable. For example:

```
orig_airport NOT IN ('ABQ', 'AKL', 'DSM')
```

becomes

```
orig_airport <> 'ABQ'
AND orig_airport <> 'AKL'
AND orig_airport <> 'DSM'
```

In addition, large lists are sorted in ascending order for performance reasons.

OR Transformations

If all the OR predicates in a WHERE clause are of the form

```
simple column reference = Expression
```

where the *columnReference* is the same for all predicates in the OR chain, Cloudscape transforms the OR chain into an IN list of the following form:

```
simple column reference IN (Expression1, Expression2, ...,
ExpressionN)
```

The new predicate may be optimizable.

For example, Cloudscape can transform the following statement:

```
SELECT * FROM Flights
WHERE flight_id = 'AA1111'
OR flight_id = 'US5555'
OR flight_id = ?
```

into this one:

```
SELECT * FROM Flights
WHERE flight_id IN ('AA1111', 'US5555', ?)
```

If this transformed IN list is a static IN list, Cloudscape also performs the static IN list transformation (see “Static IN Predicate Transformations” on page A-5).

Transitive Closure

The transitive property of numbers states that if A = B and B = C, then A = C.

Cloudscape applies this property to query predicates to add additional predicates to the query in order to give the optimizer more information. This process is called *transitive closure*. There are two types of transitive closure:

- transitive closure on join clauses

  Applied first, if applicable
- transitive closure on search clauses

Transitive Closure on Join Clauses

When a join statement selects from three or more tables, Cloudscape analyzes any equijoin predicates between simple column references within each query block and adds additional equijoin predicates where possible if they do not currently exist. For example, Cloudscape transforms the following query:

```
SELECT * FROM Groups, HotelBookings, FlightBookings
WHERE Groups.group_id = HotelBookings.group_id
AND HotelBookings.group_id = FlightBookings.group_id
```

into the following:

```
SELECT * FROM Groups, HotelBookings, FlightBookings
WHERE Groups.group_id = HotelBookings.group_id
AND HotelBookings.group_id = FlightBookings.group_id
AND Groups.group_id = FlightBookings.group_id
```

On the other hand, the optimizer knows that one of these equijoin predicates is redundant and will throw out the one that is least useful for optimization.

Transitive Closure on Search Clauses

Cloudscape applies transitive closure on search clauses after transitive closure on join clauses. For each sargable predicate where a simple column reference is compared with a constant (or the IS NULL and IS NOT NULL operators), Cloudscape looks for an equijoin predicate between the simple column reference and a simple column reference from another table in the same query block. For each such equijoin predicate, Cloudscape then searches for a similar comparison (the

same operator) between the column from the other table and the same constant. Cloudscape adds a new predicate if no such predicate is found.

Cloudscape performs all other possible transformations on the predicates (described in “Predicate Transformations” on page A-2) before applying transitive closure on search clauses.

For example, given the following statement:

```
SELECT * FROM HotelBookings, FlightBookings
WHERE HotelBookings.group_id = FlightBookings.group_id
AND HotelBookings.group_id BETWEEN 1 AND 3
AND HotelBookings.group_id <> 2
AND FlightBookings.group_id <> 4
```

Cloudscape first performs any other transformations:

- the BETWEEN transformation on the second predicate:

```
AND HotelBookings.group_id >= 1
AND HotelBookings.group_id <= 3
```

Cloudscape then performs the transitive closure:

```
SELECT * FROM HotelBookings, FlightBookings
WHERE HotelBookings.group_id = FlightBookings.group_id
AND HotelBookings.group_id >= 1
AND HotelBookings.group_id <= 3
AND HotelBookings.group_id <> 2
AND HotelBookings.group_id <> 4
AND FlightBookings.group_id >= 1
AND FlightBookings.group_id <= 3
AND FlightBookings.group_id <> 2
AND FlightBookings.group_id <> 4
```

When a sargable predicate uses the = operator, Cloudscape can remove all equijoin predicates comparing that column reference to another simple column reference from the same query block as part of applying transitive closure, because the equijoin predicate is now redundant, whether or not a new predicate was added. For example:

```
SELECT * FROM HotelBookings, FlightBookings
WHERE HotelBookings.group_id = FlightBookings.group_id
AND HotelBookings.group_id = 1
```

becomes (and is equivalent to)

```
SELECT * FROM HotelBookings, FlightBookings
WHERE FlightBookings.group_id = 1
AND HotelBookings.group_id = 1
```

The elimination of redundant predicates gives the optimizer more accurate selectivity information and improves performance at execution time.

View Transformations

When Cloudscape evaluates a statement that references a view, it transforms the reference to a view into a derived table. It may make additional transformations to improve performance.

- "View Flattening" on page A-9
- "Predicates Pushed into Views or Derived Tables" on page A-10
- "Materialization" on page A-11

View Flattening

When evaluating a statement that references a view, Cloudscape internally transforms a view into a derived table. This derived table may also be a candidate for *flattening* into the outer query block.

A view or derived table can be flattened into the outer query block if all of the following conditions are met:

- The select list is composed entirely of simple column references and constants.
- There is no GROUP BY clause in the view.
- There is no DISTINCT in the view.

For example, given view *v1(a,b):*

```
SELECT Hotels.hotel_name, Cities.city_id
FROM Hotels, Cities
WHERE Hotels.city_id = Cities.city_id
```

and a SELECT that references it:

```
SELECT a, b
FROM v1 WHERE a = 'Hotel du Quai Voltaire'
```

after the view is transformed into a derived table, the internal query is

```
SELECT a, b
FROM (select Hotels.hotel_name, Cities.city_id
FROM Hotels, Cities
WHERE Hotels.city_id = Cities.city_id) v1(a, b)
WHERE a = 'Hotel du Quai Voltaire'
```

After view flattening it becomes

```
SELECT Hotels.hotel_name as a, Cities.city_id AS b
FROM Hotels, Cities
WHERE Hotels.city_id = Cities.city_id
AND Hotels.hotel_name = 'Hotel du Quai Voltaire'
```

Predicates Pushed into Views or Derived Tables

An SQL-J statement that references a view may also include a predicate. Consider the view *v2 (a,b)*:

```
CREATE VIEW v2(a,b) AS
SELECT city_id, MAX(normal_rate)
FROM Hotels
GROUP BY city_id
```

The following statement references the view and includes a predicate:

```
SELECT *
FROM v2
WHERE a = 2
```

When Cloudscape transforms that statement by first transforming the view into a derived table, it places the predicate at the top level of the new query, outside the scope of the derived table:

```
SELECT a, b
FROM (SELECT city_id, MAX(normal_rate) FROM Hotels GROUP BY
city_id) v2(a, b)
WHERE a = 2
```

In the example in the preceding section (see "View Flattening" on page A-9), Cloudscape was able to flatten the derived table into the main SELECT, so the predicate in the outer SELECT could be evaluated at a useful point in the query. This is not possible in this example, because the underlying view does not satisfy all the requirements of view flattening.

However, if the source of all of the column references in a predicate is a simple column reference in the underlying view or table, Cloudscape is able to *push* the predicate *down* to the underlying view. Pushing down means that the qualification described by the predicate can be evaluated when evaluating the view is being evaluated, which is more efficient. In our example, the column reference in the outer predicate, *a*, in the underlying view is a simple column reference to the underlying base table. So the final transformation of this statement after predicate push-down is

```
SELECT a, b
FROM (SELECT city_id, MAX(normal_rate) FROM Hotels WHERE
city_id = 2 GROUP BY city_id) v1(a, b)
```

Without the transformation, Cloudscape would have to scan the entire table *t1* to form all the groups, only to throw out all but one of the groups. With the transformation, Cloudscape is able to make that qualification part of the derived table.

If there were a predicate that referenced column *b*, it could not be pushed down, because in the underlying view, column *b* is not a simple column reference.

Predicate push-down transformation includes predicates that reference multiple tables from an underlying join.

Subquery Processing and Transformations

Subqueries are notoriously expensive to evaluate. This section describes some of the transformations that Cloudscape makes internally to reduce the cost of evaluating them.

- "Materialization" on page A-11
- "Flattening a Subquery into a Normal Join" on page A-12
- "Flattening a Subquery into an EXISTS Join" on page A-15

Materialization

Materialization means that a subquery is evaluated only once. A subquery can be materialized if it is a noncorrelated expression subquery. A correlated subquery is one that references columns in the outer query, and so has to be evaluated for each row in the outer query.

For example:

```
SELECT * FROM Hotels WHERE hotel_id = (SELECT MAX(hotel_id)
FROM HotelAvailability)
```

In this statement, the subquery needs to be evaluated only once.

This type of subquery must return only one row. If evaluating the subquery causes a cardinality violation (if it returns more than one row), an exception will be thrown at the beginning of execution.

Subquery materialization is detected prior to optimization, which allows the optimizer to see a materialized subquery as an unknown constant value. The comparison is thus optimizable.

In other words, the original statement is transformed into the following two statements:

```
constant = SELECT MAX(hotel_id) FROM HotelAvailability

SELECT * FROM hotels
WHERE hotel_id = constant
```

The second statement is optimizable.

Flattening a Subquery into a Normal Join

Subqueries are allowed to return more than one row when used with IN, EXISTS, and ANY. However, for each row returned in the outer row, Cloudscape evaluates the subquery until it returns one row—it doesn't evaluate the subquery for all rows returned.

For example, given two tables, *t1* and *t2*:

c1
1
2
3

c1
2
2
1

and the following query:

```
SELECT c1 FROM t1 WHERE c1 IN (SELECT c1 FROM t2)
```

the results would be

```
1
2
```

Simply selecting *t1.c1* when simply joining those tables has different results:

```
SELECT t1.c1 FROM t1, t2 WHERE t1.c1 = t2.c1

1
2
2
```

Statements that include such subqueries can be flattened into joins only if the subquery does not introduce any duplicates into the result set (in our example, the subquery introduced a duplicate and so cannot simply be flattened into a join). If this requirement and other requirements (listed below) are met, however, the statement is flattened such that the tables in the subquery's FROM list are treated as if they were inner to the tables in the outer FROM list.

For example, our query could have been flattened into a join if *c2* in *t2* had a unique index on it. It wouldn't have introduced any duplicate values into the result set.

The requirements for flattening into a normal join are:

- The subquery is not under an OR.
- The subquery type is EXISTS, IN, or ANY, or it is an expression subquery on the right side of a comparison operator.
- The subquery is not in the SELECT list of the outer query block.
- There are no aggregates in the SELECT list of the subquery.
- The subquery does not have a GROUP BY clause.
- There is a uniqueness condition that ensures that the subquery does not introduce any duplicates if it is flattened into the outer query block.
- Each table in the subquery's FROM list (after any view, derived table, or subquery flattening) must be a base table.
- If there is a WHERE clause in the subquery, there is at least one table in the subquery whose columns are in equality predicates with expressions that do not include any column references from the subquery block. These columns must be a superset of the key columns for any unique index on the table. For all other tables in the subquery, the columns in equality

predicates with expressions that do not include columns from the same table are a superset of the unique columns for any unique index on the table.

Flattening into a normal join gives the optimizer more options for choosing the best query plan. For example, if the following statement:

```
SELECT huge.* FROM huge
WHERE c1 IN (SELECT c1 FROM tiny)
```

can be flattened into

```
SELECT huge.* FROM huge, tiny WHERE huge.c1 = tiny.c1
```

the optimizer can choose a query plan that will scan *tiny* and do a few probes into the huge table instead of scanning the huge table and doing a large number of probes into the tiny table.

Here is an expansion of the example used earlier in this section. Given

```
CREATE TABLE t1 (c1 INT)

CREATE TABLE t2 (c1 INT PRIMARY KEY)

CREATE TABLE t3 (c1 INT PRIMARY KEY)

INSERT INTO t1 VALUES (1), (2), (3)

INSERT INTO t2 VALUES (1), (2), (3)

INSERT INTO t3 VALUES (2), (3), (4)
```

this query

```
SELECT t1.* FROM t1 WHERE t1.c1 IN
    (SELECT t2.c1 FROM t2, t3 WHERE t2.c1 = t3.c1)
```

should return the following results:

```
2
3
```

The query satisfies all the requirements for flattening into a join, and the statement can be transformed into the following one:

```
SELECT t1.*
FROM t1, t2, t3
WHERE t1.c1 = t2.c1
AND t2.c1 = t3.c1
AND t1.c1 = t3.c1
```

The following query:

```
SELECT t1.*
FROM t1 WHERE EXISTS
(SELECT * FROM t2, t3 WHERE t2.c1 = t3.c1 AND t2.c1 = 3)
```

can be transformed into

```
SELECT t1.*
FROM t1, t2, t3
WHERE t1.c1 = t2.c1
AND t2.c1 = t3.c1
AND t1.c1 = t3.c1
```

Flattening a Subquery into an EXISTS Join

An EXISTS join is a join in which the right side of the join needs to be probed only once for each outer row. Using such a definition, an EXISTS join does not literally use the EXISTS keyword. Cloudscape treats a statement as an EXISTS join when there will be at most one matching row from the right side of the join for a given row in the outer table.

A subquery that cannot be flattened into a normal join because of a uniqueness condition can be flattened into an EXISTS join if it meets all the requirements (see below). Do you remember the first example in the previous section (“Flattening a Subquery into a Normal Join” on page A-12)?

```
SELECT c1 FROM t1 WHERE c1 IN (SELECT c1 FROM t2)
```

This query could not be flattened into a normal join because such a join would return the wrong results. However, this query can be flattened into a join recognized internally by the Cloudscape system as an EXISTS join. When processing an EXISTS join, Cloudscape knows to stop processing the right side of the join after a single row is returned. The transformed statement would look something like this:

```
SELECT c1 FROM t1, t2
WHERE t1.c1 = t2.c1
EXISTS JOIN INTERNAL SYNTAX
```

Requirements for flattening into an EXISTS join:

- The subquery is not under an OR.
- The subquery type is EXISTS, IN, or ANY.
- The subquery is not in the SELECT list of the outer query block.
- There are no aggregates in the SELECT list of the subquery.
- The subquery does not have a GROUP BY clause.

- The subquery has a single entry in its FROM list that is a base table.
- None of the predicates in the subquery, including the additional one formed between the left side of the subquery operator and the column in the subquery's SELECT list (for IN or ANY subqueries), include any subqueries, method calls, or field accesses.

When a subquery is flattened into an EXISTS join, the table from the subquery is made join-order-dependent on all the tables with which it is correlated. This means that a table must appear inner to all the tables on which it is join-order-dependent. (In subsequent releases this restrictions may be relaxed.) For example:

```
SELECT t1.* FROM t1, t2
WHERE EXISTS (SELECT * FROM t3 WHERE t1.c1 = t3.c1)
```

gets flattened into

```
SELECT t1.* FROM t1, t2, t3 WHERE t1.c1 = t3.c1
```

where *t3* is join order dependent on *t1*. This means that the possible join orders are (*t1*, *t2*, *t3*), (*t1*, *t3*, *t2*), and (*t2*, *t1*, *t3*).

DISTINCT Elimination in IN, ANY, and EXISTS Subqueries

An IN, ANY, or EXISTS subquery evaluates to true if there is at least one row that causes the subquery to evaluate to true. These semantics make a DISTINCT within an IN, ANY, or EXISTS subquery unnecessary. The following two queries are equivalent and the first is transformed into the second:

```
SELECT * FROM t1 WHERE c1 IN
    (SELECT DISTINCT c2 FROM t2 WHERE t1.c3 = t2.c4)
SELECT * FROM t1 WHERE c1 IN
    (SELECT c2 FROM t2 WHERE t1.c3 = t2.c4)
```

IN/ANY Subquery Transformation

An IN or ANY subquery that is guaranteed to return at most one row can be transformed into an equivalent expression subquery (a scalar subquery without the IN or ANY). The subquery must not be correlated. Subqueries guaranteed to return at most one row are:

- Simple VALUES clauses

- SELECTs returning a nongrouped aggregate

For example:

```
WHERE C1 IN (SELECT MIN(c1) FROM T)
```

can be transformed into

```
WHERE C1 = (SELECT MIN(c1) FROM T)
```

This transformation is considered before subquery materialization. If the transformation is performed, the subquery becomes materializable. In the example, if the IN subquery were not transformed, it would be evaluated anew for each row.

The subquery type transformation is shown in Table A-1:

Table A-1 IN or ANY Subquery Transformations for Subqueries Returning a Single Row

Before Transformation	After Transformation
`c1 IN (SELECT ...)`	`c1 = (SELECT ...)`
`c1 = ANY (SELECT ...)`	`c1 = (SELECT ...)`
`c1 <> ANY (SELECT ...)`	`c1 <> (SELECT ...)`
`c1 > ANY (SELECT ...)`	`c1 > (SELECT ...)`
`c1 >= ANY (SELECT ...)`	`c1 >= (SELECT ...)`
`c1 < ANY (SELECT ...)`	`c1 < (SELECT ...)`
`c1 <= ANY (SELECT ...)`	`c1 <= (SELECT ...)`

Sort Avoidance

Sorting is an expensive process. Cloudscape tries to eliminate unnecessary sorting steps where possible.

- "DISTINCT Elimination Based on a Uniqueness Condition" on page A-17
- "Combining ORDER BY and DISTINCT" on page A-19
- "Combining ORDER BY and UNION" on page A-20

DISTINCT Elimination Based on a Uniqueness Condition

A DISTINCT (and the corresponding sort) can be eliminated from a query if a uniqueness condition exists that ensures that no duplicate values will be returned. If no duplicate values are returned, the DISTINCT node is superfluous, and

Cloudscape transforms the statement internally into one without the DISTINCT keyword.

The requirements are:

- No GROUP BY list.
- SELECT list contains at least one simple column reference.
- Every simple column reference is from the same table.
- Every table in the FROM list is a base table.
- *Primary table*
  There is at least one unique index on one table in the FROM list for which *all* the columns appear in one of the following:
  - equality predicates with expressions that do not include any column references
  - simple column references in the SELECT list
- *Secondary table*
  All the other tables in the FROM list also have at least one unique index for which all the columns appear in one of the following:
  - equality predicates with expressions that do not include columns from the same table
  - simple column references in the SELECT list

For example:

```
CREATE TABLE tab1 (c1 INT,
    c2 INT,
    c3 INT,
    c4 CHAR(2),
    PRIMARY KEY (c1, c2, c3))

CREATE TABLE tab2 (c1 INT,
    c2 INT,
    PRIMARY KEY (c1, c2))

INSERT INTO tab1 VALUES (1, 2, 3, 'WA'),
    (1, 2, 5, 'WA'),
    (1, 2, 4, 'CA'),
    (1, 3, 5, 'CA'),
    (2, 3, 1, 'CA')

INSERT INTO tab2 VALUES (1, 2),
    (1, 3),
    (2, 2),
    (2, 3)
```

```
-- all the columns in the index on the only table appear
-- in the way required for the Primary table
SELECT DISTINCT c1, c2, c3, c4
FROM tab1

-- all the columns in the index on the only table appear
-- in the way required for the Primary table
SELECT DISTINCT c3, c4
FROM tab1
WHERE c1 = 1
AND c2 = 2
AND c4 = 'WA'

-- all the columns in the index on tab1 appear
-- in the way required for the Primary table,
-- and all the columns in the
-- other tables appear in the way required
-- for a Secondary table
SELECT DISTINCT tab1.c1, tab1.c3, tab1.c4
FROM tab1, tab2
WHERE tab1.c2 = 2
AND tab2.c2 = tab1.c2
AND tab2.c1 = tab1.c1
```

Combining ORDER BY and DISTINCT

Without a transformation, a statement that contains both DISTINCT and ORDER BY would require two separate sorting steps—one to satisfy DISTINCT and one to satisfy ORDER BY. (Currently, Cloudscape uses sorting to evaluate DISTINCT. There are, in theory, other ways to accomplish this.) In some situations, Cloudscape can transform the statement internally into one that contains only one of these keywords. The requirements are:

- The columns in the ORDER BY list must be a subset of the columns in the SELECT list.
- All the columns in the ORDER BY list are sorted in ascending order.

A unique index is not required.

For example:

```
SELECT DISTINCT miles, meal
FROM Flights
ORDER BY meal
```

is transformed into

```
SELECT DISTINCT miles, meal
FROM Flights
```

Combining ORDER BY and UNION

Without a transformation, a statement that contains both ORDER BY and UNION would require two separate sorting steps—one to satisfy ORDER BY and one to satisfy UNION. (Currently Cloudscape uses sorting to eliminate duplicates from a UNION.)

In some situations, Cloudscape can transform the statement internally into one that contains only one of these keywords (the ORDER BY is thrown out). The requirements are:

- The columns in the ORDER BY list must be a subset of the columns in the select list of the left side of the union.
- All he columns in the ORDER BY list must be sorted in ascending order and they must be an in-order prefix of the columns in the target list of the left side of the UNION.

Cloudscape will be able to transform the following statements:

```
SELECT miles, meal
FROM Flights
UNION VALUES (1000, 'D')
ORDER BY miles
```

```
SELECT city_id, tour_level FROM Hotels
UNION
SELECT city_id, tour_level FROM Groups
ORDER BY Hotels.city_id, Hotels.tour_level
```

```
SELECT city_id, tour_level FROM Hotels
UNION
SELECT city_id, tour_level FROM Groups
ORDER BY city_id, tour_level
```

Cloudscape cannot avoid two sorting nodes in the following statement, because of the order of the columns in the ORDER BY clause:

```
SELECT city_id, tour_level FROM Hotels
UNION
SELECT city_id, tour_level FROM Groups
ORDER BY tour_level, city_id
```

Aggregate Processing

COUNT(nonNullableColumn)

Cloudscape transforms COUNT(nonnullable column) into COUNT(*). This improves performance by potentially reducing the number of referenced columns in the table (each referenced column needs to be read in for each row) and by giving the optimizer more access path choices. For example, the cheapest access path for

```
SELECT COUNT(*) FROM t1
```

is the index on *t1* with the smallest number of leaf pages, and the optimizer is free to choose that path.

Index

A

Access path
  how optimizer chooses 4-14
  overriding optimizer 4-25, 6-5, 6-8, 6-8
  when optimizer's cost estimates for it are accurate 4-14
  when optimizer's cost estimates for it are less accurate 4-15
Application design
  performance implications 3-1
Application performance
  analyzing 3-12
Authentication
  turning on 5-15
Authentication provider
  specifying 5-11
Automatic stale plan invalidation 3-5

B

Base table
  definition A-2
BETWEEN transformations A-3
Booting
  configuring system 5-72
Bulk fetch 4-24
  configuring system-wide default 5-31
  how optimizer chooses 4-24
  overriding system default 4-26, 6-3
  when useful 6-4
bulkFetch optimizer-override property 6-3

C

Caches
  performance benefits of priming 2-5
Checkpoint 3-12
Class loading
  enabling 5-17
  how to minimize impact of 3-10
  tuning 2-8
  when it occurs in Cloudscape 3-10
Class path
  performance implications of 2-10
Cloudscape properties
  setting 1-4–1-12
cloudscape.authentication.ldap.searchAuthDN 5-5
cloudscape.authentication.ldap.searchAuthPW 5-7
cloudscape.authentication.ldap.searchbase 5-8
cloudscape.authentication.ldap.searchfilter 5-9
cloudscape.authentication.provider 5-11
cloudscape.authentication.server 5-13
cloudscape.connection.requireAuthentication 5-15
cloudscape.database.classpath 5-17
cloudscape.database.defaultConnectionMode 5-19
cloudscape.database.forceDatabaseLock 5-21
cloudscape.database.fullAccessUsers 5-22
cloudscape.database.noAutoBoot 2-8, 5-24
cloudscape.database.propertiesOnly 1-3, 5-25
cloudscape.database.readOnlyAccessUsers 5-26
cloudscape.infolog.append 5-28
cloudscape.jdbc.metadataStoredPreparedStatements 5-29
cloudscape.language.bulkFetchDefault 5-31
cloudscape.language.defaultIsolationLevel 5-33
cloudscape.language.logStatementText 5-35
cloudscape.language.preloadClasses 5-36
cloudscape.language.spsCacheSize 5-37
cloudscape.language.stalePlanCheckInterval 5-39

cloudscape.language.statementCacheSize 2-10, 5-41
cloudscape.locks.deadlockTimeout 5-45
cloudscape.locks.deadlockTrace 5-47
cloudscape.locks.escalationThreshold 4-24, 5-48
cloudscape.locks.monitor 5-50
cloudscape.locks.waitTimeout 5-51
cloudscape.properties file 1-4, 1-6
cloudscape.service 5-53
cloudscape.storage.initialPages 5-55
cloudscape.storage.minimumRecordSize 5-57
cloudscape.storage.pageCacheSize 5-59
cloudscape.storage.pageReservedSpace 5-60
cloudscape.storage.pageSize 2-5, 5-62
cloudscape.storage.rowLocking 5-64
cloudscape.storage.tempDirectory 5-66
cloudscape.stream.error.field 5-68
cloudscape.stream.error.file 5-69
cloudscape.stream.error.logSeverityLevel 5-70
cloudscape.stream.error.method 5-71
cloudscape.system.bootAll 2-8, 5-72
cloudscape.system.home 5-73
cloudscape.system.home
  determining location of *cloudscape.properties* 1-6
cloudscape.user 5-74
Compilation
  avoiding by using *PreparedStatements* 3-8
  avoiding by using stored prepared statements 3-8
  performance hit 3-8
Compilation classes
  loading at boot time 5-36
COM.cloudscape.database.Database 1-7
COM.cloudscape.database.PropertyInfo 1-9
COM.cloudscape.vti.VTICosting 4-27
Configuring Cloudscape 1-4—1-12
Conglomerate-specific properties 5-1
  setting 1-8
  verifying 1-9
constraint optimizer-override property 6-5
Covering indexes 4-7
Cursors
  affected by indexes 4-11

D

Data page cache
  performance benefits of increasing 2-4
Database design
  performance implications 3-1
Database pages
  preallocating 5-55
DatabaseMetaData
  performance tip 2-11
Databases
  booting all in system 5-72
  booting upon connection only 5-24
  external to system directory 5-53
Database-wide properties
  protecting against overrides 5-25
  protecting for embedded environment 1-3
  setting 1-7
  shortcut for setting 1-8
  verifying value of 1-7
Deadlocks
  monitoring 5-50
  timeout property 5-45
  tracing 5-47
  wait timeout property 5-51
Derived table
  definition A-2
DISTINCT
  combined with ORDER BY A-19
  eliminated for uniqueness condition A-17
Dynamic properties 1-4

E

Equality predicate
  definition A-2
Equijoin predicate
  definition A-2
Equijoins
  optimizable 4-13
Error log
  configuring severity of errors that appear in 5-70
  redirecting to a stream 5-68, 5-71
  specifying file name 5-69
EXISTS join
  definition A-15
Expensive queries
  how to avoid 3-4
  importance of avoiding 3-4

F

FLOAT
  java.sql type converted to DOUBLE PRECISION when retrieved 2-8

H

Hash join strategy 4-13, 6-10
Hash joins
  requirements for 4-13

use of memory 4-13

I

index optimizer-override property 6-8
Index use
  analyzing 3-3, 3-12
  how optimizer chooses 4-14
Indexes
  cost of maintaining 4-10
  definition 4-2
  how they work 4-1—4-11
  performance benefits of 2-4, 3-2
  querying system tables to get system-generated name 1-9
  when they are useful 4-4
Information log
  overwriting 5-28
Internal transformation of statements 4-6
  BETWEEN predicates A-3
  COUNT A-21
  LIKE predicates A-3
  OR predicates A-6
  predicates A-2—A-6
  Sort avoidance A-17
  static IN predicates A-5
  transitive closure A-7
  views A-9
IN/ANY subquery transformation A-16
Isolation levels
  configuring default for system or database 5-33

J

java.sql.ResultSet.getXXX methods
  performance implications of 2-7
JDBC *DatabaseMetaData*
  improving performance of 5-29
JIT
  performance benefits of 2-2
Join order
  analyzing 3-12
  how optimizer chooses 4-16
  overriding optimizer 4-25
  performance implications of 3-3, 4-11
  rules of thumb 4-12
Join strategies 4-13
  how optimizer chooses 4-18
  overriding optimizer 4-26
joinOrder optimizer-override property 6-9
Joins
  optimizability of 4-7
  performance factors 4-11—4-14
joinStrategy optimizer-override property 6-10
JVMs (Java Virtual Machines)
  importance for performance 2-2

L

Language transformations for performance A-1—A-21
Large database pages
  requirement for more memory 2-7
LDAP
  configuring 5-5, 5-7, 5-8, 5-9, 5-13
LIKE transformations A-3
Lock escalation 5-48
  at runtime (for statement) 4-24
  at runtime (for transaction) 4-28
  by user 4-30
  chosen by optimizer 4-21
  threshold 4-24
Lock granularity
  how optimizer chooses 4-21
  tuning 4-26
LOCK TABLE statement 4-30
Locking
  performance implications of 4-28
Locks
  monitoring 5-50
LONGVARBINARY
  java.sql type converted to BIT VARYING when retrieved 2-8
LONGVARCHAR
  java.sql type converted to VARCHAR when retrieved 2-8

M

Matching index scans
  definition 4-5
MAX() optimization 4-32
Memory
  allocating more memory to an application 2-7
MIN() optimization 4-32

N

Nested loop join strategy 4-13, 6-10
NIS+
  configuring 5-13

O

Optimizable
  definition 4-5, A-2
Optimizable equijoins
  definition 4-13
Optimizable operators 4-6

Optimization 4-1—4-26
  requirements for 4-5
Optimizer
  accuracy of 4-27, 4-27
  correcting accuracy of by forcing table scans 4-27
  decisions made by 4-14
  description 4-1
  overriding 4-25, 6-1—6-11
Optimizer overrides 6-1—6-11
  scope of 6-1
OR transformations A-6
ORDER BY
  cost-based avoidance of 4-19

P

Page cache size
  configuring 5-59
Page size
  configuring 5-62
  performance implications of 2-5
  performance trade-offs of large pages 2-6
Performance
  key issues 3-1
Performance of statements
  analyzing 3-18—3-25
Platforms
  recommended 2-2
Predicates
  definition A-2
  directly optimizable 4-5
  indirectly optimizable 4-6
  pushed down into views A-10
PreparedStatements
  performance benefits of 3-8
Properties
  conglomerate-specific scope 1-2, 5-1
  database-wide scope 1-2
  dynamic vs. static 1-4
  implications of having various ways to set 1-11
  in client/server mode 1-8, 1-10
  persistence of 1-2
  precedence of 1-3
  publishing 1-8
  scope of 1-2
  setting using a *Properties* object 1-5
  system-wide scope 1-2, 1-4
  verifying 1-7
  verifying for tables and indexes 1-9

Q

Qualifiers
  in matching index scans 4-9
Query optimization 4-1—4-11

R

Recompiling
  when advisable 2-9
Reserving space on a page 5-60
Row-level locking
  disabling 5-64
RunTimeStatistics
  overview 3-19
  using 3-19

S

Sargable
  definition A-2
Selectivity assumptions of optimizer 4-15
Shutdown
  performance benefits of 2-10
Simple column reference
  definition A-2
Sort avoidance 4-18, A-17
Stale statements 3-4
  definition 3-7
Statement cache
  performance benefits of 2-10, 3-9
Statement execution
  analyzing 3-12
  debugging 5-35
Statement execution plan 3-21
Statement plans
  automatic invalidation of 5-39
Statements
  and stale statement execution plans 3-6
  caching 5-41
  debugging 5-35
  when they are optimizable 4-5
Static IN transformations A-5
Static properties 1-4
  unable to set through SQL-J statement 1-5
Statistics timing attribute
  use of 3-20
Storage properties 1-9, 5-1
Stored prepared statement cache 5-37
Stored prepared statements 3-8
  caching of across multiple connections 3-15
  need to recompile 2-9, 3-14
  performance benefits of 2-3, 3-13
  performance hit caused by recompilation 3-17

sample values for parameters for optimization 3-14
staleness of requiring recompilation 3-18
what makes them invalid 3-17
Subqueries
elimination of DISTINCT in IN, ANY, and EXISTS subqueries A-16
flattening of A-12
flattening of to an EXISTS join A-15
materialization of A-11
processing and transformations of A-11
System properties
setting in a *Properties* object 1-5
setting on command line 1-5
System-wide properties
setting 1-6

T

Table-level locking
forcing for a specific table at creation time 4-22
tmp directory
setting location for 5-66
Transitive closure A-7

U

UNION
avoiding ordering during A-20
User authentication
turning on 5-15
User authorization
configuring 5-19, 5-22, 5-26
Users
creating 5-74

V

Verifying database-wide properties 1-7
Verifying properties used in tables and indexes 1-9
View flattening A-9
View transformations A-9
VTIs
and costing information 4-27