# Table of Contents

**SQL-J Grammar Index**

**About This Document**
- Purpose of This Document
- Audience
- How This Document Is Organized

## 1 SQL-J Language Reference
- Capitalization and Special Characters
- SQL-J Identifiers
  - Rules for Both *SQL92Identifiers* and *JavaIdentifiers*
  - *SQL92Identifier*
    - Qualifying Dictionary Objects
    - *SchemaName*
    - *TableName*
    - *ViewName*
    - *CorrelationName*
    - *ColumnName*
    - *SimpleColumnName*
    - *IndexName*
    - *ConstraintName*
    - *CursorName*
    - *MethodAlias*
    - *ClassAlias*
AggregateName 1-10
StatementName 1-10
TriggerName 1-11
AuthorizationIdentifier 1-11
JarName 1-11
JavaIdentifier 1-12
JavaClassName 1-12
JavaMethodName 1-13
JavaFieldName 1-13
Statements 1-15
Interaction with the Dependency System 1-15
Interaction with the Java Data Types 1-17
ALTER STATEMENT statement 1-18
ALTER TABLE statement 1-20
AT ISOLATION clause 1-24
CALL statement 1-25
CONSTRAINT clause 1-29
CREATE AGGREGATE statement 1-38
CREATE CLASS ALIAS statement 1-39
CREATE INDEX statement 1-41
CREATE METHOD ALIAS statement 1-43
CREATE SCHEMA statement 1-44
CREATE STATEMENT statement 1-45
CREATE TABLE statement 1-48
CREATE TRIGGER statement 1-51
CREATE VIEW statement 1-60
CROSS JOIN 1-62
DELETE statement 1-63
DROP AGGREGATE statement 1-64
DROP CLASS ALIAS statement 1-65
DROP INDEX statement 1-66
DROP METHOD ALIAS statement 1-67
DROP SCHEMA statement 1-68
DROP STATEMENT statement 1-69
DROP TABLE statement 1-70
DROP TRIGGER statement 1-71
DROP VIEW statement 1-72
EXECUTE STATEMENT statement 1-73
ExternalVirtualTable 1-76
FOR UPDATE clause 1-78
FROM clause 1-79
GROUP BY clause 1-80
HAVING clause 1-82
Data Types

Built-In Type Overview

SQL-J Types, Java Types, and JDBC Types

Numeric Types

Numeric Type Overview
Numeric Type Promotion in Expressions
Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type
Scale for Decimal Arithmetic

Implicit Type Conversion of String Types

Implicit Conversions from Strings to Other Built-In Data Types
Implicit Conversion to Strings from Other Built-In Data Types
Limitations on Implicit Conversions from or to String Types

Comparing Booleans with Numeric Types and Assigning Numeric Types to Booleans

Data Types and Comparison, Sorting, and Ordering

BIT
BIT VARYING
BOOLEAN
CHAR
DATE
DECIMAL
DOUBLE PRECISION
FLOAT
INTEGER
LONG VARBINARY, LONG BIT VARYING, LONG BINARY
LONG VARCHAR
LONGINT
NUMERIC

Cloudscape Reference Manual
### SQL-J Expressions

- **Expression Precedence**
  - 1-221
- **Boolean expression**
  - 1-223
- **Dynamic Parameters**
  - 1-226

#### Java Expressions

- **Method Invocation**
  - 1-231
- **Field Access**
  - 1-238
- **INSTANCEOF Expression**
  - 1-240
- **NEW**
  - 1-241
- **GETCURRENTCONNECTION()**
  - 1-242

#### SQL-J and Java Type Correspondence

- **Column Values and Type Correspondence**
  - 1-243
- **SQL-J to Java Type Correspondence**
  - 1-244
- **Java to SQL-J Type Correspondence**
  - 1-245
- **Method Resolution and Type Correspondence**
  - 1-246

### 2 SQL-J Keywords and Reserved Words

- **Complete List of Reserved Words**
  - 2-1
- **Nonreserved Keywords**
  - 2-4

### 3 Cloudscape 3.0 Support for SQL-92 Features

- **SYSALIASES**
  - 4-3
- **SYSCHECKS**
  - 4-5
- **SYSCOLUMNS**
  - 4-6
- **SYSCONGLOMERATES**
  - 4-9
- **SYSCONSTRAINTS**
  - 4-11
- **SYSDEPENDS**
  - 4-13
- **SYSErrORS**
  - 4-15
- **SYSFILES**
  - 4-17
- **SYSFORIGNKEYS**
  - 4-18
- **SYSDBCTYPEINFO**
  - 4-19
- **SYSKEYS**
  - 4-20
5 Cloudscape Exception Messages and SQL States 5-1

6 JDBC Reference 6-1
Core JDK 1.1.x java.sql Classes, Interfaces, and Methods 6-2
  java.sql.Driver 6-3
  java.sql.DriverManager.getConnection 6-5
  java.sql.Driver.getPropertyInfo 6-10
  java.sql.Connection 6-11
  java.sql.DatabaseMetaData 6-12
  java.sql.Statement 6-14
  java.sql.PreparedStatement 6-15
  java.sql.CallableStatement 6-17
  java.sql.ResultSet 6-20
  java.sql.ResultSetMetaData 6-22
  java.sql.SQLException 6-23
  java.sql.SQLWarning 6-24
  java.sql.Types 6-25

JDBC 2.0-Only Features 6-26
  java.sql.Connection 6-27
  java.sql.ResultSet 6-28
  java.sql.Statement 6-29
  java.sql.PreparedStatement 6-30
  java.sql.CallableStatement 6-31
  java.sql.DatabaseMetaData 6-32
  java.sql.ResultSetMetaData 6-33
  java.sql.BatchUpdateException 6-34

JDBC Escape Syntax 6-35
  CALL 6-36
d 6-37
oj 6-38
t 6-39
ts 6-40

7 Database Connection URL Attributes 7-1
  autocommit=false 7-2
  bootPassword=key 7-3
  create=true 7-4
  current=true 7-5
databaseName=nameofDatabase  7-6
dataEncryption=true  7-7
logDevice=logDirectoryPath  7-8
password=userPassword  7-9
shutdown=true  7-10
upgrade=true  7-12
user=userName  7-13
(no attributes)  7-14

8 J2EE Compliance: Java Transaction API and javax.sql Extensions  8-1

J2EE Overview  8-1
Requirements  8-2
The JTA API  8-3
Notes on Product Behavior  8-3
Recovered Global Transactions  8-3
XAConnections and User Names and Passwords  8-3

javax.sql: JDBC Extensions  8-5

Appendix A

Cloudscape API  A-1
Stand-Alone Tools and Utilities  A-1
Core Classes  A-2
Boot Classes  A-2
Data Source Classes  A-3
SQL-J Extensions  A-3
Types Used in System Tables  A-4
Other Types  A-4
SQLException-Related  A-5
User-Throwable  A-5
Interfaces for User-Defined Aggregates  A-6
VTIs  A-6
Miscellaneous Utilities and Interfaces  A-7

Index  IN-1

Cloudscape Version 3.0
SQL-J Grammar
Index

A
ALTER STATEMENT statement 1-18
ALTER TABLE statement 1-20
AT ISOLATION clause 1-24
AVG (aggregate function) 1-131

B
BIT (data type) 1-190
BIT VARYING (data type) 1-192
BIT_LENGTH (function) 1-133
BOOLEAN (data type) 1-193
Boolean expression 1-223

C
CALL statement 1-25
CAST (function) 1-134
CHAR (data type) 1-194
CHAR_LENGTH (function) 1-143
CHARACTER_LENGTH (function) 1-143
Concatenation (function) 1-144
Conditional (?:) (expression) 1-145
CONSTRAINT clause 1-29
COUNT (*) (aggregate function) 1-147
COUNT (aggregate function) 1-146
CREATE AGGREGATE statement 1-38
CREATE CLASS ALIAS statement 1-39
CREATE INDEX statement 1-41
CREATE METHOD ALIAS statement 1-43
CREATE SCHEMA statement 1-44
CREATE STATEMENT statement 1-45
CREATE TABLE statement 1-48
CREATE TRIGGER statement 1-51
CREATE VIEW statement 1-60
CROSS JOIN (expression) 1-62
CURRENT_DATE (function) 1-148
CURRENT_TIME (function) 1-149
CURRENT_TIMESTAMP (function) 1-150
CURRENT_USER (function) 1-151

DATE (data type) 1-195
DECIMAL (data type) 1-196
DELETE statement 1-63
DISTINCT clause (part of SelectExpression) 1-102
DOUBLE PRECISION (data type) 1-198
DROP AGGREGATE statement 1-64
DROP CLASS ALIAS statement 1-65
DROP INDEX statement 1-66
DROP METHOD ALIAS statement 1-67
DROP SCHEMA statement 1-68
DROP STATEMENT statement 1-69
DROP TABLE statement 1-70
DROP TRIGGER statement 1-71
DROP VIEW statement 1-72

EXECUTE STATEMENT statement 1-73
ExternalVirtualTable 1-76
EXTRACT (function) 1-152

Field Access (expression) 1-238
FLOAT (data type) 1-199
FOR UPDATE clause 1-78
FROM clause 1-79

GetCurrentConnection() (function) 1-242
GROUP BY clause 1-80

HAVING clause 1-82

INNER JOIN (expression) 1-83
INSERT statement 1-86
INSTANCEOF expressions 1-240
INTEGER (data type) 1-200

Java Data Types 1-211
JOIN Operation (expression) 1-89

LEFT OUTER JOIN (expression) 1-90
LOCK TABLE statement 1-91
LONG BINARY (data type) 1-201
LONG BIT VARYING (data type) 1-201
LONG VARBINARY (data type) 1-201
LONG VARCHAR (data type) 1-202
LONGINT (data type) 1-203
LOWER (function) 1-153
LTRIM (function) 1-154

M
MAX (aggregate function) 1-155
Method Invocation (expression) 1-231
MIN (aggregate function) 1-157

N
NEW (expression) 1-241
NUMERIC (data type) 1-204

O
OCTET_LENGTH (function) 1-159
ORDER BY clause 1-93

P
PROPERTIES clause 1-95

R
REAL (data type) 1-205
RIGHT OUTER JOIN (expression) 1-100
RTRIM (function) 1-160
RUNTIMESTATISTICS() built-in function 1-161

S
ScalarSubquery (expression) 1-101
SELECT statement 1-106
SelectExpression 1-102
SESSION_USER (function) 1-169
SET CONSTRAINTS statement 1-109
SET RUNTIMESTATISTICS statement 1-111
SET SCHEMA statement 1-112
SET STATISTICS TIMING statement 1-113
SET TRANSACTION ISOLATION LEVEL statement 1-114
SET TRIGGERS statement 1-115
SMALLINT (data type) 1-206
SUBSTR (function) 1-172
SUBSTRING (function) 1-170
SUM (aggregate function) 1-174

T
TableExpression (expression) 1-116
TableSubquery (expression) 1-119
TIME (data type) 1-207
TIMESTAMP (data type) 1-208
TINYINT (data type) 1-209
TRIM (function) 1-175

U
UNION operator (part of Query) 1-98
SQL-J Grammar Index

UPDATE statement 1-121
UPPER (function) 1-177
USER (function) 1-178
User-Defined Data Types 1-211
User-DefinedAggregate 1-179

V
VALUES expression 1-123
VARCHAR (data type) 1-210

W
WHERE clause 1-125
WHERE CURRENT OF clause 1-127
About This Document

• “Purpose of This Document” on page xv
• “Audience” on page xv
• “How This Document Is Organized” on page xvi

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, the Cloudscape Reference Manual, provides reference information about Cloudscape™. It covers Cloudscape’s SQL-J language, the Cloudscape implementation of JDBC™, Cloudscape system catalogs, Cloudscape error messages, Cloudscape properties, and SQL-J keywords.

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.

How This Document Is Organized

This document includes the following chapters:

- **Chapter 1, “SQL-J Language Reference”**
  Reference information about Cloudscape’s SQL-J language, including manual pages for statements, functions, and other syntax elements.

- **Chapter 2, “SQL-J Keywords and Reserved Words”**
  SQL-J keywords beyond the standard SQL-92 keywords.

- **Chapter 3, “Cloudscape 3.0 Support for SQL-92 Features”**
  A list of SQL-92 features that Cloudscape does and does not support.

- **Chapter 4, “Cloudscape System Tables”**
  Reference information about the Cloudscape system catalogs.

- **Chapter 5, “Cloudscape Exception Messages and SQL States”**
  Information about Cloudscape exception messages.

- **Chapter 6, “JDBC Reference”**
  Information about Cloudscape’s implementation of the JDBC interface include support for JDBC 2.0 features.

- **Chapter 7, “Database Connection URL Attributes”**
  Information about the supported attributes to Cloudscape’s JDBC database connection URL.

- **Chapter 8, “J2EE Compliance: Java Transaction API and javax.sql Extensions”**
  Information about the supported attributes to Cloudscape’s support for the Java Transaction API.

- **Appendix A, “Cloudscape API”**
  Notes about proprietary APIs for Cloudscape.
1 SQL-J Language Reference

The Cloudscape language, called SQL-J, is a Java-object extension to SQL-92. Version 3.0 implements an SQL-92 core subset and Java object support.

This document provides an overview of the current SQL-J language by describing the statements, built-in functions, data types, expressions, and special characters it contains. It includes the following sections:

- “Capitalization and Special Characters” on page 1-2
- “SQL-J Identifiers” on page 1-3
- “Statements” on page 1-15
- “Built-In Functions” on page 1-128
- “Data Types” on page 1-180
- “SQL-J Expressions” on page 1-217
- “SQL-J and Java Type Correspondence” on page 1-243
Capitalization and Special Characters

Using the classes and methods of JDBC, you submit SQL-J statements to Cloudscape as strings. The character set permitted for the strings containing SQL-J statements is ASCII, with support for Unicode escapes. Within these strings, the following rules apply:

- Double quotation marks delimit special identifiers referred to in SQL-92 as delimited identifiers.
- Single quotation marks delimit character strings.
- Within a character string, to represent a single quotation mark or apostrophe, use two single quotation marks. To represent a double quotation mark, use a double quotation mark (which requires a backslash escape character within a Java program).

```
VALUES 'Joe''s umbrella'
VALUES 'He said, "hello!"
```

```java
n = stmt.executeUpdate(
    "UPDATE aTable setStringcol = 'He said, "hello!"');
```

- SQL-J keywords are case-insensitive. For example, you can type the keyword SELECT as SELECT, Select, select, or sELECT.
- SQL-92-style identifiers are case-insensitive (see "SQL92Identifier" on page 1-4), unless they are delimited.
- Java-style identifiers are always case-sensitive (see “JavaIdentifier” on page 1-12).
- * is a wildcard within a SelectExpression. See “The * Wildcard” on page 1-103. It can also be the multiplication operator. In all other cases, it is a syntactical metasymbol that flags items you can repeat 0 or more times.
- % and _ are character wildcards when used within character strings following a LIKE operator. See “LIKE” in Table 1-8 on page 1-224.
- Two dashes (--) and a newline character delimit a comment, as per the SQL-92 standard. The two dashes start the comment and the newline character ends the comment.

**NOTE:** Within SQL-J statements, Cloudscape supports only Unicode escapes, not Unicode strings. However, you can store Unicode strings in text columns. To do so, use a dynamic parameter within a prepared statement. Use the setString method to set the value of the string to the Unicode string before executing the prepared statement.
SQL-J Identifiers

An identifier is the representation within the language of items created by the user, as opposed to language keywords or commands. SQL-J has two kinds of identifiers:

- Some identifiers stand for dictionary objects, which are the objects you create—such as tables, views, indexes, columns, constraints, and stored prepared statements—that are stored in a database. They are called dictionary objects because Cloudscape stores information about them in the system tables, sometimes known as a data dictionary. SQL-92 also defines ways to alias these objects within certain statements.
- Other identifiers stand for user constructs recognized by a JVM, such as Java classes, methods, packages, and fields.

Each kind of identifier must conform to a different set of rules. Identifiers representing dictionary objects must conform to SQL-92 identifier rules and are thus called SQL92Identifiers. Identifiers representing Java constructs must conform to rules that enable them to be recognized by both Cloudscape and the JVM and are thus called JavaIdentifiers.

Rules for Both SQL92Identifiers and JavaIdentifiers

Non-delimited identifiers are identifiers not surrounded by double quotation marks. Delimited identifiers are identifiers surrounded by double quotation marks.

A non-delimited identifier must begin with a letter and contain only letters, underscore characters (_), and digits.

A delimited identifier can contain any characters within the quotation marks. The enclosing quotation marks are not part of the identifier; they serve only to mark its beginning and end. Spaces at the end of a delimited identifier are insignificant (truncated). Cloudscape translates two consecutive double quotation marks within a delimited identifier as one double quotation mark—that is, the “translated” double quotation mark becomes a character in the delimited identifier.

Periods within delimited identifiers are not separators but are part of the identifier (the name of the dictionary object or Java construct being represented).

So, in the following example:

"A.B"

is a dictionary object or Java construct, while

"A" . "B"
is a dictionary object qualified by another dictionary object (such as a column named “B” within the table “A”) or a Java construct qualified by another Java construct (Java class “B” within the Java package “A”).

**SQL92Identifier**

An SQL92Identifier is a dictionary object identifier that conforms to the rules of SQL-92. SQL-92 states that identifiers for dictionary objects are limited to 128 characters and are case-insensitive (unless delimited), because they are automatically translated into uppercase by the system. You cannot use reserved words as identifiers for dictionary objects unless they are delimitated. If you attempt to use a name longer than 128 characters, SQLException X0X11 is raised.

Cloudscape defines keywords beyond those specified by the SQL-92 standard (see Chapter 2, “SQL-J Keywords and Reserved Words”).

**Example**

```sql
-- the view name is stored in the
-- system catalogs as ANIDENTIFIER
CREATE VIEW AnIdentifier AS VALUES 1

-- the view name is stored in the system
-- catalogs with case intact
CREATE VIEW "ACaseSensitiveIdentifier" AS VALUES 1
```

This section describes the rules for using SQL92Identifiers to represent the following dictionary objects:

- **SchemaName**
- **TableName**
- **CorrelationName**
- **ColumnName**
- **SimpleColumnName**
- **IndexName**
- **ConstraintName**
- **CursorName**
- **MethodAlias**
- **ClassAlias**
- **AggregateName**
• StatementName
• TriggerName
• AuthorizationIdentifier
• JarName

**Qualifying Dictionary Objects**

Since some dictionary objects can be contained within other objects, you can qualify those dictionary object names. Each component is separated from the next by a period. An SQL92Identifier is “dot-separated.” You qualify a dictionary object name in order to avoid ambiguity.

**SchemaName**

A SchemaName represents a schema. Schemas contain other dictionary objects, such as tables and indexes. Schemas provide a way to name a subset of tables and other dictionary objects within a database.

You can explicitly create or drop a schema. The default user schema is the APP schema (if no user name is specified at connection time). You are not allowed to create dictionary objects in the SYS schema.

Thus, you can qualify references to tables with the schema name. When a schema name is not specified, the default schema name is implicitly inserted. System tables are placed in the SYS schema. You must qualify all references to system tables with the SYS schema identifier. (For more information about system tables, see Chapter 4, “Cloudscape System Tables”.)

A schema is hierarchically the highest level of dictionary object, so you cannot qualify a schema name.

**Syntax**

SQL92Identifier

**Example**

-- APP.CustomizedTours is a TableName qualified by a SchemaName
SELECT COUNT(*) FROM APP.CustomizedTours

-- You must qualify system catalog names with their schema, SYS
SELECT COUNT(*) FROM SYS.SysColumns
TableName

A TableName represents a table. You can qualify a TableName with a SchemaName.

Syntax

[SchemaName.] SQL92Identifier

Example

-- APP.FlightBookings is a TableName that includes a SchemaName
SELECT COUNT(*) FROM APP.FlightBookings

ViewName

A ViewName represents a table or a view. You can qualify a ViewName with a SchemaName.

Syntax

[SchemaName.] SQL92Identifier

Example

-- This is a View qualified by a SchemaName
SELECT COUNT(*) FROM APP.Segments_SeatBookings

CorrelationName

A CorrelationName is given to a table expression in a FROM clause as a new name or alias for that table. You do not qualify a CorrelationName with a SchemaName.

Syntax

SQL92Identifier

Example

-- C is a CorrelationName
SELECT C.region
FROM APP.Countries C
**ColumnName**

In many places in the SQL-J syntax, you can represent the name of a column by qualifying it with a `TableName` or `CorrelationName`.

In some situations, you cannot qualify a `ColumnName` with a `TableName` or a `CorrelationName`, but must use a `SimpleColumnName` instead. Those situations are:

- creating a table (CREATE TABLE statement)
- specifying updatable columns in a cursor
- in a column’s correlation name in a SELECT expression (see `SelectExpression`)
- in a column’s correlation name in a `TableExpression` (see `TableExpression`)

**Syntax**

```
[ `{ TableName | CorrelationName } . ] SQL92Identifier
```

**Example**

```sql
-- C.Country is a ColumnName qualified with a CorrelationName.
-- SELECT C.Country
-- FROM APP.Countries C
```

**SimpleColumnName**

A `SimpleColumnName` is used to represent a column when it cannot be qualified by a `TableName` or `CorrelationName`. This is the case when the qualification is fixed, as it is in a column definition within a CREATE TABLE statement, and in ORDER BY clauses.

**Syntax**

```
SQL92Identifier
```

**Example**

```sql
-- country is a SimpleColumnName
CREATE TABLE Countries
  (country VARCHAR(26) PRIMARY KEY,
```
country_ISO_code CHAR(2),
region VARCHAR(26))

**IndexName**

An IndexName represents an index. Indexes live in schemas, so you can qualify their names with SchemaNames. Indexes on system tables are in the SYS schema.

**Syntax**

```
[ SchemaName . ] SQL92Identifier
```

**Example**

```
DROP INDEX APP.OrigIndex
-- OrigIndex is an IndexName without a SchemaName
CREATE BTREE INDEX OrigIndex ON Flights(orig_airport)
```

**ConstraintName**

A ConstraintName represents a constraint. Constraints live in schemas, so you can qualify their names with SchemaNames. There are no system constraints.

**Syntax**

```
[ SchemaName . ] SQL92Identifier
```

**Example**

```
CREATE TABLE Orders
(OrderId INTEGER
CONSTRAINT OrderKey PRIMARY KEY,
OrderItem VARCHAR(40))
```

**CursorName**

A CursorName refers to a cursor. No SQL-J language command exists to assign a name to a cursor. Instead, you use the JDBC API to assign names to cursors or to retrieve system-generated names. For more information, see “Naming or Accessing the Name of a Cursor” on page 6-9 in the Cloudscape Developer’s Guide. If you assign a name to a cursor, you can refer to that name from within SQL-J statements.
You cannot qualify a *CursorName*.

**Syntax**

```
SQL92Identifier
```

**Example**

```
UPDATE SET OrderItem = 'ACME Widget'
WHERE CURRENT OF OrderRow
```

**MethodAlias**

A *MethodAlias* is used to refer to a static Java method. It allows you to execute the method without having to enter the full class and method name. Before using a method alias, you must explicitly define it in the current database with a CREATE METHOD ALIAS statement. Method aliases are *SQL92Identifiers*, so they are case-insensitive. Method aliases do not live in schemas, so they are not qualified with schema names. See “CREATE METHOD ALIAS statement” on page 1-43 for more information.

**Syntax**

```
SQL92Identifier
```

**Example**

```
-- abs is a user-defined method alias for java.lang.Math.abs
SELECT abs(running_total)
FROM Groups

-- Method aliases are case-insensitive
SELECT ABS(running_total)
FROM Groups
```

**ClassAlias**

Cloudscape allows you to create a *ClassAlias* for a Java class. Once you create an alias for a class, you do not have to use the complete package and class name for the class, you use the alias instead. In addition, you do not have to use the CLASS keyword in the situations when it would be required for a *JavaClassName*.
Syntax

Example

--- HotelStay is an alias for JBMSTours.serializabletypes.HotelStay
--- if it were not an alias, you would have to
--- call the method like this:
--- CALL (CLASS
JBMSTours.serializabletypes.HotelStay).archiveRecordsVTI etc...
CALL HotelStay.archiveRecordsVTI(
gGetCurrentConnection(), CURRENT_DATE)

AggregateName

Cloudscape allows you to create user-defined aggregates. These are used in the same places as SQL’s built-in aggregates such as MIN and MAX.

Syntax

Example

--- MaxButOne is a user-defined aggregate
SELECT MAXBUTOne(miles) FROM Flights

StatementName

A StatementName refers to a stored prepared statement named by a user. Stored prepared statements can be qualified with schema names. See “CREATE STATEMENT statement” on page 1-45 for more information about stored prepared statements.

Syntax

Cloudscape Reference Manual
Example

-- create a stored prepared statement in the current schema
CREATE STATEMENT queryFlights
AS SELECT * FROM Flights WHERE orig_airport = ?

TriggerName

A TriggerName refers to a trigger created by a user.

Syntax

[ SchemaName . ] SQL92Identifier

Example

DROP TRIGGER HotelsCascadingDelete

AuthorizationIdentifier

User names within the Cloudscape system are known as authorization identifiers. The authorization identifier represents the name of the user, if one has been provided in the connection request. If a schema name equal to the authorization identifier exists, the user defaults to that schema. User names may be case-sensitive within the authentication system, but they are always case-insensitive within Cloudscape’s authorization system unless they are delimited. For more information, see “Users and Authorization Identifiers” on page 8-21 in the Cloudscape Developer’s Guide.

Syntax

SQL92Identifier

CALL PropertyInfo.setDatabaseProperty(
   'cloudscape.database.fullAccessUsers', '"!Amber",FRED')

JarName

When you store application logic in a jar file within a Cloudscape database, you give it a JarName within the Cloudscape system. You qualify jar names with schema names.
CALL PropertyInfo.setDatabaseProperty(
    'cloudscape.database.classpath',
    'APP.ToursLogic:APP.AccountingLogic')

JavaIdentifier

A JavaIdentifier represents a Java class, object, method, field name, or property. JavaIdentifiers are case-sensitive whether they are delimited identifiers or not. They are not automatically translated to uppercase. There is no upper limit to the length of a JavaIdentifier.

Example

-- these two identifiers are equivalent
java.lang.String
"java"."lang"."String"

This section details the rules for the ways JavaIdentifiers represent the following Java constructs:

- JavaClassName
- JavaMethodName
- JavaFieldName

JavaClassName

A JavaClassName refers to a Java class or interface. The name must be fully qualified with the package, even if it is in the java.lang package.

The Java class must be declared to be public and must be in the class path of the JVM in which Cloudscape is running (see “Deploying Java Classes for Use as Java Data Types” on page 3-6 in the Cloudscape Developer’s Guide).

Syntax

[ JavaIdentifier . ]* JavaIdentifier

Examples

VALUES NEW java.lang.Integer('3')
CREATE TABLE Cities
  (city SERIALIZE(JBMSTours.serializabletypes.City))

CALL (CLASS
  JBMSTours.serializabletypes.HotelStay).archiveRecordsVTI(
    getCurrentConnection(), current_date)

JavaMethodName

A JavaMethodName refers to a Java method. It is attached to an expression identifying the Java class or object on which the method exists.

The Java method must be declared to be public and its class and must be in the class path of the JVM in which Cloudscape is running (see “Deploying Java Classes for Use as Java Data Types” on page 3-6 in the Cloudscape Developer’s Guide). The class defining the method need not itself be declared public; i.e., the usual Java accessibility rules apply.

The method can be final or non-final. The expression in which the method is used determines whether the method can be static or non-static.

**Syntax**

```
JavaIdentifier
```

**Example**

```
SELECT DISTINCT city.getName()
FROM Cities
WHERE city->city_id = 5
```

JavaFieldName

A JavaFieldName refers to a Java field. It is attached to an expression identifying the Java class or object on which the field exists.

The Java field must be declared to be public and its class must be in the class path of the JVM in which Cloudscape is running (see “Deploying Java Classes for Use as Java Data Types” on page 3-6 in the Cloudscape Developer’s Guide). The class defining the field need not itself be declared public; that is, the usual Java accessibility rules apply.

The field can be final or non-final. The expression in which the method is used determines whether the field can be static or non-static.
Syntax

JavaIdentifier

Example

-- name is a JavaFieldName
SELECT City->name
FROM Cities
This section provides manual pages for both high-level language constructs and parts thereof. For example, the CREATE INDEX statement is a high-level statement that you can execute directly via the JDBC interface. This section also includes clauses, which are not high-level statements and which you cannot execute directly but only as part of a high-level statement. The ORDER BY and WHERE clauses are examples of this kind of clause. Finally, this section also includes some syntactically complex portions of statements called expressions, for example `SelectExpression` and `TableSubquery`. These clauses and expressions receive their own manual pages for ease of reference.

Unless it is explicitly stated otherwise, you can execute or prepare and then execute all the high-level statements, which are all marked with the word `statement`, via the interfaces provided by JDBC. This manual indicates whether an expression can be executed as a high-level statement.

**NOTE:** You cannot execute DDL in a target database in a replication system. DDL stands for data definition language and consists of statements that create or drop dictionary objects. Examples are the CREATE TABLE statement, the DROP SCHEMA statement, and the ALTER STATEMENT statement.

### Interaction with the Dependency System

Cloudscape internally tracks the dependencies of prepared statements and stored prepared statements, which are SQL-J statements that are precompiled before being executed. Typically they are prepared (precompiled) once and executed multiple times.

Prepared statements and stored prepared statements depend on the tables, indexes, views, aliases, work units (dictionary objects), and statements they reference. Removing or modifying the dictionary objects or statements on which they depend invalidates them internally, which means that Cloudscape will automatically try to recompile the statement when you execute it. If the statement fails to recompile, the execution request fails. However, if you take some action to restore the broken dependency (such as restoring the missing table), you can execute the same prepared statement, because Cloudscape will recompile it automatically at the next execute request.

Statements depend on one another—an UPDATE WHERE CURRENT statement depends on the statement it references. Removing the statement on which it depends invalidates the UPDATE WHERE CURRENT statement.
In addition, prepared statements prevent execution of certain DDL statements if there are open results sets on them.

Manual pages for each statement detail what actions would invalidate that statement, if prepared.

Here is an example using the Cloudscape tool ij:

```
ij> CREATE TABLE mytable (mycol INT);
0 rows inserted/updated/deleted
ij> INSERT INTO mytable VALUES (1), (2), (3);
3 rows inserted/updated/deleted
ij> -- this example uses the ij command prepare,
-- which prepares a statement
prepare p1 AS 'INSERT INTO MyTable VALUES (4)';
ij> -- p1 depends on mytable;
execute p1;
1 row inserted/updated/deleted
ij> -- Cloudscape executes it without recompiling
CREATE INDEX i1 ON mytable(mycol);
0 rows inserted/updated/deleted
ij> -- p1 is temporarily invalidated because of new index
execute p1;
1 row inserted/updated/deleted
ij> -- Cloudscape automatically recompiles p1 and executes it
DROP TABLE mytable;
0 rows inserted/updated/deleted
ij> -- Cloudscape permits you to drop table
-- because result set of p1 is closed
-- however, the statement p1 is temporarily invalidated
CREATE TABLE mytable (mycol INT);
0 rows inserted/updated/deleted
ij> INSERT INTO mytable VALUES (1), (2), (3);
3 rows inserted/updated/deleted
ij> execute p1;
1 row inserted/updated/deleted
ij> -- Because p1 is invalid, Cloudscape tries to recompile it
-- before executing.
-- It is successful and executes.
DROP TABLE mytable;
0 rows inserted/updated/deleted
ij> -- statement p1 is now invalid,
```
-- and this time the attempt to recompile it
-- upon execution will fail
execute p1;
ERROR 42X05: Table ‘MYTABLE’ does not exist.

**Interaction with the Java Data Types**

In the current release, there is no dependency checking of Java data types. That is, if a prepared statement uses a Java class, and that class changes or disappears, the dependency system does not detect the change, and the prepared statement is not invalidated. This could cause the prepared statement to get an error or to produce unexpected results. Consider the following example:

```sql
CREATE TABLE Cities
  (City SERIALIZE(JBMSTours.serializabletypes.City))

SELECT City.getName()
FROM Cities
```

If the class `JBMSTours.serializabletypes.City` changes after the SELECT statement is compiled, but before it is executed, the dependency system does not detect that the class has changed, and the SELECT statement could get an error or unexpected results.
ALTER STATEMENT statement

The ALTER STATEMENT statement allows you to recompile a stored prepared statement or statements created with a CREATE STATEMENT statement.

Cloudscape allows you to recompile statements explicitly with the ALTER STATEMENT RECOMPILE statement for:

- a particular statement
  by specifying the statement name
- all statements in all schemas
  by specifying RECOMPILE ALL
- all invalid statements in all schemas
  by specifying RECOMPILE INVALID

NEW: The ability to recompile all invalid statements is new in Version 3.0.

The USING clause allows you to provide new sample parameters to the ALTER STATEMENT statement for use in optimization (see “CREATE STATEMENT statement” on page 1-45). The statement is recompiled when the ALTER STATEMENT statement is issued.

You cannot change the text of the stored prepared statement with this statement. Instead, you must drop the statement and re-create it.

Syntax

```sql
{  
  ALTER STATEMENT StatementName RECOMPILE  
  [ USING SingleRowResultSet ] |  
  ALTER STATEMENT RECOMPILE { ALL | INVALID }  
}
```

Typically, a stored prepared statement is recompiled as needed when a dependency changes. For example, a prepared statement that depends on an index that is dropped is marked invalid and thus automatically recompiled the next time the statement is executed. Force recompilation for all invalid statements instead of waiting for them to be executed before deploying an application onto read-only media.

In addition, there are some times when you may want to explicitly force a statement to be recompiled, even when it is not invalid, such as when a statement may have gone stale or when a new index is created that will help the performance of an existing prepared statement, and the statement is a SELECT. (For information about
stale statements, see “Stale Stored Prepared Statements” on page 3-18 Tuning Cloudscape).

The USING clause serves the same purpose that it serves in the CREATE STATEMENT statement; it gives default values to the optimizer for best optimization. See “About the USING Clause” on page 1-45.

**Examples**

```
-- Recompile a statement. The recompilation is done the next
-- time the statement is executed
ALTER STATEMENT getFullFlightInfo RECOMPILE

-- Recompile the number_of_rooms_taken statement with
-- some new sample values. Do the recompilation now.
ALTER STATEMENT getFullFlightInfo
RECOMPILE USING VALUES ('US1473', 2)

-- recompile all invalid statements
ALTER STATEMENT RECOMPILE INVALID
```
SQL-J Language Reference

ALTER TABLE statement

The ALTER TABLE statement allows you to:

- add a column to a table
- add a constraint to a table
- drop an existing constraint from a table
- add a default value for a column in a table
- drop a default value for a column in a table by setting the default to null
- override row-level locking for the table (or drop the override)

**NEW:** The ability to add or drop a default value for a column in the table and to override row-level locking for the table are new features in Version 3.0.

Syntax

```
ALTER TABLE TableName
{
  ADD COLUMN ColumnDefinition  |
  ADD CONSTRAINT clause  |
  DROP CONSTRAINT ConstraintName [ RESTRICT | CASCADE ] |
  MODIFY { ( ColumnDefault  ) | ColumnDefault  } |
  SET LOCKING = { TABLE | ROW }
}
```

**ColumnDefault**

```
ColumnName DEFAULT { ConstantExpression | NULL }
```

ALTER TABLE does not affect any view that references the table being altered. This includes views that have an "*" in their SELECT list. You must drop and re-create those views if you wish them to return the new columns.

Adding Columns

The syntax for the `ColumnDefinition` for a new column is the same as for a column in a CREATE TABLE statement. This means that a column constraint can be placed on the new column within the ALTER TABLE ADD COLUMN statement. However, a column with a NOT NULL constraint can be added to an existing table only if the table is empty; otherwise, an exception is thrown when the ALTER TABLE statement is executed.

(See “Adding Constraints” on page 1-21 for the other limitations.) Just as in CREATE TABLE, if the column definition includes a primary key constraint, the
column will be made non-nullable, so an exception is thrown if you attempt to add a primary key column to a table that is not empty.

**NOTE:** If a table has an UPDATE trigger without an explicit column list, adding a column to that table in effect adds that column to the implicit update column list upon which the trigger is defined, and all references to transition variables are invalidated so that they pick up the new column.

### Adding Constraints

**ALTER TABLE** ADD CONSTRAINT adds a table-level constraint to an existing table. Any supported table-level constraint type can be added via **ALTER TABLE**. The following limitations exist on adding a constraint to an existing table:

- All columns included in a primary key must be non-nullable.

**NOTE:** When creating a primary key in a CREATE TABLE or **ALTER TABLE** ADD COLUMN statement, you automatically make all columns in a primary key non-nullable. **ALTER TABLE** ADD CONSTRAINT does not do this, so the columns it references when defining a primary key constraint must already be NOT NULL.

- When adding a foreign key or check constraint to an existing table, Cloudscape checks the table to make sure existing rows satisfy the constraint. If any row is invalid, Cloudscape throws a statement exception and the constraint is not added.

For information on the syntax of constraints, see “CONSTRAINT clause” on page 1-29. Use the syntax for table-level constraint when adding a constraint with the **ADD TABLE** ADD CONSTRAINT syntax.

### Dropping Constraints

**ALTER TABLE** DROP CONSTRAINT drops a constraint on an existing table. To drop an unnamed constraint, you must specify the generated constraint name stored in **SYS.SYSCONSTRAINTS** as a delimited identifier.

Dropping a primary key, unique, or foreign key constraint drops its backing index. When you drop a primary key or unique constraint and there may be foreign key constraints referencing that primary key or unique constraint, you have two options:

- **Specifying CASCADE** means Cloudscape drops all referencing foreign key constraints and their backing indexes.
Specifying RESTRICT means that Cloudscape throws an exception if there are any foreign keys referencing the primary key that you want to drop. The constraint is not dropped. This is the default behavior.

### Changing the Lock Granularity for the Table

The SET LOCKING clause allows you to override row-level locking for the specific table, if your system is set for row-level locking. (If your system is set for table-level locking, you cannot change the locking granularity to row-level locking, although Cloudscape allows you to use the SET LOCKING clause in such a situation without throwing an exception.) To override row-level locking for the specific table, set locking for the table to TABLE. If you created the table with table-level locking granularity, you can change locking back to ROW with the SET LOCKING clause in the ALTER TABLE statement. For information about why this is sometimes useful, see “About the Optimizer’s Selection of Lock Granularity” on page 4-21 in *Tuning Cloudscape*.

### Adding or Dropping Defaults

You can specify a default value for a column. A default value is the value that is inserted into a column if no other value is specified. If not explicitly specified, the default value of a column is NULL. If you add a default to a new column, existing rows in the table gain the default value in the new column. If you add a default to an existing column, existing rows in the table do not gain the default value in the new column.

To drop a default, set the default value to NULL. Doing so does not affect the values of the column for existing rows.

For more information about defaults, see “CREATE TABLE statement” on page 1-48.

### Examples

```sql
-- Add a new column with a column-level constraint
-- to an existing table
-- An exception will be thrown if the table
-- contains more than 1 row
-- since the newcol will be initialized to NULL
-- in all existing rows in the table
ALTER TABLE Cities
ADD COLUMN Region VARCHAR(26)
CONSTRAINT new_constraint NOT NULL
```
-- Add a new unique constraint to an existing table
-- An exception will be thrown if duplicate keys are found
ALTER TABLE Countries
ADD CONSTRAINT new_unique UNIQUE(country)

-- add a new foreign key constraint to the
-- Hotels table. Each row in Hotels is checked
-- to make sure it satisfied the contraints.
-- if any rows don’t satisfy the constraint, the
-- constraint is not added
ALTER TABLE HOTELS ADD CONSTRAINT City_FK
    FOREIGN KEY (city_id) REFERENCES Cities

-- Drop a primary key constraint from a table
-- You will not be able to drop this constraint if
-- there are any foreign keys referencing it
ALTER TABLE Cities DROP CONSTRAINT Cities_PK

-- Drop a primary key constraint from a table
-- and all referencing foreign keys
ALTER TABLE Cities DROP CONSTRAINT Cities_PK CASCADE

-- add a default value to a column
-- (existing rows are not affected)
ALTER TABLE HotelAvailability
MODIFY rooms_taken DEFAULT 1

-- drop a default value for a column
ALTER TABLE HotelAvailability
MODIFY rooms_taken DEFAULT NULL

Dependency System

An ALTER TABLE statement causes all statements that are dependent on the table
being altered to be recompiled before their next execution. ALTER TABLE is not
allowed if there are any open cursors that reference the table being altered.
**AT ISOLATION clause**

Allows a user to specify an isolation level for a SELECT statement without committing the current transaction. This clause changes the isolation level for the current statement only, not the transaction.

The other ways of changing the isolation level for a connection, JDBC method calls and the SET TRANSACTION ISOLATION LEVEL statement, commit the current transaction.

For information about isolation levels, see “Locking, Concurrency, and Isolation” on page 6-13 in the *Cloudscape Developer’s Guide*.

**Syntax**

```
AT ISOLATION { READ COMMITTED | SERIALIZABLE }
```

**Examples**

```sql
-- set the isolation level to SERIALIZABLE for this
-- statement only
SELECT *
FROM Flights
WHERE flight_id BETWEEN 'AA1111' AND 'AA1112'
AT ISOLATION SERIALIZABLE
```
CALL statement

The CALL statement executes a public method call associated with a Java class (a static method) or an object (a non-static method). The CALL statement is the only way to execute methods with void return types.

The \( ? = \) syntax allows you to retrieve a return value from the method call. It is provided for use within a JDBC CallableStatement; however, it also works from a Statement or PreparedStatement. In Cloudscape, it is more typical to use the VALUES expression to retrieve a return value from a method call. The following two statements are similar:

```
-- this statement returns a single-column, single-row
-- ResultSet
VALUES city.findCity(getCurrentConnection(), 35)
```

```
-- this statement does not return a result set
-- but it does return a value through an OUT parameter
=? CALL city.findCity(getCurrentConnection(), 35)
```

The method can return a void type or any valid Java type. If its return type is not void, the returned value is simply ignored unless the \( ? = \) syntax is used. If the \( ? = \) syntax is used, the method must return a value.

To iterate a method call over a number of rows, use a SelectExpression. The CALL statements operates on only a single parameter set at a time.

Syntax

\[
[\ ? = \ ] \text{CALL} \\
\{ \\
\quad \text{Method Invocation} \mid \\
\quad \text{Work Unit Invocation} \\
\}
\]

CALL Statement Usage

You can use a CALL statement in all of the following cases:

- to execute a static method that does not return a value, the most typical use

  ```java
  CALL HotelStay.archiveRecords(
    DATE '1998-02-01'
  )
  ```

- to execute a static method that does return a value

  ```java
  ? = CALL City.findCity(getCurrentConnection(), CAST (? AS INT))
  ```

  \( (? = \text{CALL could be replaced by VALUES in this instance.}) \)
• to execute a method alias (an alias for a static method of a class)

CALL cleanOutFlightAvailability(getCurrentConnection())

If the method returns a value that you want to retrieve, use a VALUES expression or the ? = syntax.

• to execute a method associated with an object not serialized in the database (the object is instantiated within the statement)

CALL NEW JBMSTours.inserters.InsertCountries().confirmInsert(getCurrentConnection())

• to execute a non-static method of an object serialized in the database

CALL (SELECT customized_tour
    FROM CustomizedTours WHERE group_id = 1).printTourInfo(getCurrentConnection())

You must use a scalar subquery to limit the number of objects on which the method is called to one.

• to execute a static method of an object serialized in the database

CALL (SELECT customized_tour.getStay1()
    FROM CustomizedTours
    WHERE group_id = 1).archiveRecords(DATE’1998-02-01’)

You must use a scalar subquery to limit the number of objects on which the method is called to one. You do not use a WHERE clause. Note that it makes more sense to execute the static method off the class, not off the object. The above example is a lot simpler when executed off the class:

CALL (CLASS
    JBMSTours.serializatetypes.HotelStay).archiveRecords(DATE’1998-02-01’)

• in a Cloudsync system, to execute a work unit, a type of method alias registered with the system in which only the method call and target parameter values are replicated, not any underlying statements

CALL myWorkUnit(getCurrentConnection(), ?, ?)

The parameters are applied at the target when CALL is applied at the target, and they are saved and sent to the source for use during the next refresh. For more information about work unit invocation, see the Cloudscape Synchronization Guide.
For information on method resolution, see “Method Resolution and Type Correspondence” on page 1-246.

The CALL Statement and JDBC

Although the CALL statement works with Statements, PreparedStatements, and CallableStatements, you must use the latter if you want to use JDBC OUT or INOUT parameters. (See Chapter 6, “JDBC Reference”.)

Dependency System

The CALL statement depends on all the tables named in the Java expression, if one is used, and on all aliases used in the query. Dropping an alias invalidates a prepared CALL statement if the statement uses the alias. A DROP WORK UNIT statement invalidates a statement that uses the work unit.

There is no tracking of dependencies on any Java classes referenced in the CALL statement.

Modification of Java Object Parameters

Parameters to a CALL statement are Java objects. In embedded mode, if a Java object is passed in and modified within the method, changes to it are visible to the invoker of the method because the application and Cloudscape share the same JVM. The same holds true for any method, not just those executed with a CALL statement.

This is not true when running in client/server mode because a client and a server do not share the same JVM. (To see changed values in client/server mode, use INOUT parameters as described in Chapter 6, “JDBC Reference”.)

Consider the following example method, which alters the value of a City object parameter:

```java
public static void alterMe(City acity) throws SQLException {
    acity.name = "Altered Name";
}
```

An application creates a City object, then executes that method within a CALL statement, passing in that City object as a parameter:

```java
s = conn.createStatement();
City mycity = new City(1, "Amsterdam", "Dutch", "NL", "AMS");
/* the toString() method shows the value of the name field, which
```
should be "Amsterdam" at this point
System.out.println(mycity.toString());
PreparedStatement ps = conn.prepareStatement("CALL (CLASS JBMSTours.City).alterMe(?)");
ps.setObject(1, mycity);
ps.execute();
System.out.println(mycity.toString());

Here is the output in embedded mode:

Loaded the embedded JDBC driver
Amsterdam, Netherlands
Altered Name, Netherlands

Here is the output in client/server mode:

Loaded the client JDBC driver
Amsterdam, Netherlands
Amsterdam, Netherlands

In embedded mode, the value of the “name” field of the mycity variable was changed in the application’s memory. In client/server mode, the value of the “name” field was not changed.
A CONSTRAINT clause is an optional part of a CREATE TABLE statement or ALTER TABLE statement. A constraint is a rule to which data must conform. Constraint names are optional.

A CONSTRAINT can be one of the following:

- a column-level constraint
  Column-level constraints refer to a single column in the table and do not specify a column name (except check constraints). They refer to the column that they follow.

- a table-level constraint
  Table-level constraints refer to one or more columns in the table. Table-level constraints specify the names of the columns to which they apply. Table-level CHECK constraints can refer to 0 or more columns in the table.

Column constraints include:

- NOT NULL
  Specifies that this column cannot hold NULL values (constraints of this type are not nameable).

- NULL
  The opposite of NOT NULL (not really a constraint), it specifies that the column can hold NULL values. Specifying NULL is the same as saying nothing at all, except when the column is included in a PRIMARY KEY constraint.

- PRIMARY KEY
  Specifies the column that uniquely identifies a row in the table. Also gives the column an implicit NOT NULL constraint (in a CREATE TABLE statement, not in an ALTER TABLE statement).

- UNIQUE
  Specifies that values in the column must be unique. A NULL value is allowed.

- FOREIGN KEY
  Specifies that the values in the column must correspond to values in a referenced primary key or unique key column or that they are NULL.

- CHECK
  Specifies rules for values in the column.

Table constraints include:
• PRIMARY KEY
  Specifies the column or columns that uniquely identify a row in the table. NULL values are not allowed.

• UNIQUE
  Specifies that values in the columns must be unique.

• FOREIGN KEY
  Specifies that the values in the columns must correspond to values in referenced primary key or unique columns or that they are NULL.

NOTE: If the foreign key consists of multiple columns, and any column is NULL, the whole key is considered NULL. The insert is permitted no matter what is on the non-null columns.

• CHECK
  Specifies a wide range of rules for values in the table.

Column constraints and table constraints have the same function; the difference is in where you specify them. Table constraints allow you to specify more than one column in a PRIMARY KEY, UNIQUE, CHECK, or FOREIGN KEY constraint definition. Column-level constraints (except for check constraints) refer to only one column.

Columns that store Java objects cannot be part of a PRIMARY KEY, UNIQUE, or FOREIGN KEY constraints unless they are orderable.

NEW: Beginning in Version 3.0, you can define PRIMARY KEY, UNIQUE, and FOREIGN KEY constraints on columns that store orderable Java objects. For information, see “Indexes on Java Data Types” on page 1-42 and “Orderable Java Data Types” on page 1-214.

Syntax

Column-Level Constraint

{ NOT NULL | [CONSTRAINT ConstraintName] { CHECK (searchCondition) | { PRIMARY KEY | UNIQUE | ReferencesSpecification } [ PROPERTIES clause ] } }
### Table-Level Constraint

```sql
[CONSTRAINT ConstraintName]
{
    CHECK (searchCondition) |
    
    PRIMARY KEY (SimpleColumnName [ , SimpleColumnName ]*) |
    UNIQUE (SimpleColumnName [ , SimpleColumnName ]*) |
    FOREIGN KEY (SimpleColumnName [ , SimpleColumnName ]*) |
}

ReferencesSpecification
[
    REFERENCES TableName [ (SimpleColumnName [ , SimpleColumnName ]*) ]
]
```

#### searchCondition

A `searchCondition` is any Boolean expression that meets the requirements specified in “Requirements for Search Condition” on page 1-34.

If a `ConstraintName` is not specified, Cloudscape generates a unique constraint name (for either column or table constraints). Constraint names are not generated for NOT NULL constraints.

**NOTE:** Cloudscape recommends that you name all constraints.

### Primary Key and Unique Constraints

A primary key defines the set of columns that uniquely identifies rows in a table.

When you create a primary key, none of the columns can have NULL constraints; that is, they must not permit NULL values. When you add columns without explicit NOT NULL or NULL constraints to a primary key, they gain implicit NOT NULL constraints.

A unique constraint enforces the uniqueness of data. Unique constraints allow NULLs.

A table can have at most one PRIMARY KEY constraint, but can have multiple UNIQUE constraints.
Primary key and unique constraints are permitted on Java data types only if they are orderable (see “Orderable Java Data Types” on page 1-214).

**Foreign Key Constraints**

Foreign keys provide a way to enforce the referential integrity of a database. A foreign key is a column or group of columns within a table that references a key in some other table (or sometimes, though rarely, the same table). The foreign key must always include the columns of which the types exactly match those in the referenced primary key or unique constraint.

For a table-level foreign key constraint in which you specify the columns in the table that make up the constraint, you cannot use the same column more than once.

If there is a column list in the `ReferencesSpecification` (a list of columns in the referenced table), it must correspond either to a unique constraint or to a primary key constraint in the referenced table. The `ReferencesSpecification` can omit the column list for the referenced table if that table has a declared primary key.

If there is no column list in the `ReferencesSpecification` and the referenced table has no primary key, a statement exception is thrown. (This means that if the referenced table has only unique keys, you must include a column list in the `ReferencesSpecification`.)

Foreign key constraints are permitted on Java data types only if they are orderable (see “Orderable Java Data Types” on page 1-214).

A foreign key constraint is satisfied if there is a matching value in the referenced unique or primary key column, or if the value is NULL. If the foreign key consists of multiple columns, the foreign key value is considered NULL if any of its columns contains a NULL.

**NOTE:** It is possible for a foreign key consisting of multiple columns to allow one of the columns to contain a value for which there is no matching value in the referenced columns, per the SQL-92 standard. To avoid this situation, create NOT NULL constraints on all of the foreign key’s columns.

**Foreign Key Constraints and DML**

When you insert into or update a table with an enabled foreign key constraint, Cloudscape checks that the row does not violate the foreign key constraint by looking up the corresponding referenced key in the referenced table. If the constraint is not satisfied, Cloudscape rejects the insert or update with a statement exception.
When you update or delete a row in a table with a referenced key (a primary or unique constraint referenced by a foreign key), Cloudscape checks every foreign key constraint that references the key to make sure that the removal or modification of the row does not cause a constraint violation. If removal or modification of the row would cause a constraint violation, the update or delete is not permitted and Cloudscape throws a statement exception.

You can temporarily disable foreign key and check constraints. See “SET CONSTRAINTS statement” on page 1-109.

Cloudscape performs constraint checks at the time the statement is executed, not when the transaction commits.

**Backing Indexes**

UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints generate indexes that enforce or “back” the constraint (and are sometimes called backing indexes). UNIQUE and PRIMARY KEY constraints generate unique indexes. FOREIGN KEY constraints generate non-unique indexes. Therefore, if a column or set of columns has a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint on it, you do not need to create an index on those columns for performance. Cloudscape has already created it for you. See “Indexes and Constraints” on page 1-41.

You can specify storage properties for the indexes backing the PRIMARY KEY, UNIQUE, and FOREIGN KEY constraints with a PROPERTIES clause. A PROPERTIES clause allows you to specify page size for the backing index on a constraint and to override the values for those properties set on a database-wide basis. For backing indexes, you can specify only page size. For more information, see “PROPERTIES clause” on page 1-95.

These indexes are available to the optimizer for query optimization (see “CREATE INDEX statement” on page 1-41) and have system-generated names.

You cannot drop backing indexes with a DROP INDEX statement; you must drop the constraint or the table.

**Check Constraints**

A check constraint can be used to specify a wide range of rules for the contents of a table. A search condition (which is a boolean expression) is specified for a check constraint. This search condition must be satisfied for all rows in the table. The search condition is applied to each row that is modified on an INSERT or UPDATE at the time of the row modification. The entire statement is aborted if any check constraint is violated.
NOTE: According to the SQL-92 standard, if the boolean expression returns an UNKNOWN, it satisfies the search condition. This means that if you have a check constraint such as “c = 3”, and a row is inserted in which the value in column c is null, c=3 evaluates to UNKNOWN for that column. This means that the insert is allowed. If this situation is not acceptable to you, use the IS expression (see “IS” in Table 1-8 on page 1-225).

Requirements for Search Condition

The search condition can reference any column in the current row of the table to which the constraint is being applied. The search condition must always return the same value if applied to the same values. Thus, it cannot contain any of the following:

- Dynamic parameters (?)
- Date/Time Functions (CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP)
- RunTimeStatistics()
- Subqueries
- User-defined aggregates

About Java Methods and Fields in CHECK Constraints

Use Java methods and fields with care. They should return the same results given the same circumstances; they should not depend on anything except the arguments. For example, methods could potentially return random results. Such methods are not appropriate. It is also possible to use methods that execute SQL. Some SQL is appropriate, but some is not, because it might depend on something other than the arguments to the method (for example, data in another table). Cloudscape does not check the appropriateness of Java methods and fields in check constraints. It is up to the database designer to ensure that the use of the method or field is appropriate.

Here is an extended example showing you how seemingly harmless SQL could get you into trouble.

Subqueries are not allowed in check constraints. Suppose that is the functionality that you need. You want to check that \( t1.c1 \) is in \( t2.c1 \), and you cannot do it with a foreign key because there is no primary key or unique constraint on \( t2.c1 \). You could add a method call that contains SQL to the check constraint.

If subqueries were allowed, here’s how you would like to define the table:

Cloudscape Reference Manual
-- CREATE TABLE t1(c1 INT CHECK (c1 IN SELECT c1 FROM t2),
-- c2 INT)

This statement is not allowed. Instead, you could do the following:

-- CREATE TABLE t1(c1 INT CHECK(booleanMethod(c1)),
-- c2 INT)

where booleanMethod essentially performs the subquery.

The problem with this approach is that the contents of t2 can change independently of the contents of t1. Consider the following scenario:

```
INSERT INTO t2 (c1) VALUES 1
-- succeeds since 1 is in t2
INSERT INTO t1 VALUES (1, 1)
-- nothing to stop the delete
DELETE FROM t2 WHERE c1 = 1
-- fails since check constraint will fail
-- even though you are not changing the contents of the row
UPDATE t1 SET c2 = c2 where c1 = c2
```

Disabling Constraints

You can temporarily disable foreign key and check constraints. For example, you might want to disable such constraints before loading a lot of data. See “SET CONSTRAINTS statement” on page 1-109.

Examples

CREATE TABLE Cities
-- column-level primary key, named constraint;
(city_id INT CONSTRAINT CITIES_PK PRIMARY KEY,
city SERIALIZE(JBMSTours.serializabletypes.City),
country_ISO_code CHAR(2))

CREATE TABLE HotelAvailability (hotel_id INT ,
booking_date DATE,
rooms_taken INT,
-- the table-level primary key definition allows you to
-- include two columns in the primary key definition
PRIMARY KEY (hotel_id, booking_date))

-- Use a column-level constraint for an arithmetic check;
-- use a column-level constraint to make sure that the
-- tour level is one of the correct values (by referencing
-- static final fields), and use a table-level constraint
-- to make sure that a group’s running total does not
-- exceed its budget
-- in this example, Tour is a class alias
-- for JBMSTours.serializabletypes.Tour

CREATE TABLE Groups(
    group_id INT CONSTRAINT Groups_PK PRIMARY KEY,
    number_kids INT,
    number_adults INT,
    number_people INT CONSTRAINT people_check
                          CHECK(number_people=number_kids+number_adults),
    main_person INT,
    number_rooms INT,
    city_id INT,
    address VARCHAR(100),
    phone VARCHAR(15),
    tour_level SMALLINT CONSTRAINT level_check2
                          CHECK (tour_level IN (Tour->ECONOMYTOURLEVEL,
                                                  Tour->STANDARDTOURLEVEL,
                                                  Tour->FIRSTCLASSTOURLEVEL)),
    budget DECIMAL(11,2),
    running_total DECIMAL(11,2)
    CONSTRAINT spending_check CHECK (running_total <= budget))

-- use a check constraint to allow only appropriate
-- abbreviations for the meals

CREATE TABLE Flights
    (flight_id CHAR(6),
    segment_number INT,
    orig_airport CHAR(3),
    depart_time TIME,
    dest_airport CHAR(3),
    arrive_time TIME,
    meal CHAR(1)
    CONSTRAINT MEAL_CONSTRAINT CHECK
        (meal IN ('B', 'L', 'D', 'S')))

-- create a table whose city_id column references the
-- primary key in the Cities table
-- using a column-level foreign key constraint

CREATE TABLE hotels
    (hotel_id INT CONSTRAINT hotels_PK PRIMARY KEY,
    hotel_name VARCHAR(40) NOT NULL,
    city_id INT CONSTRAINT city_foreign_key REFERENCES Cities)
CREATE TABLE FlightAvailability
(flight_id CHAR(6),
segment_number INT,
flight_date DATE,
economy_seats_taken INT,
business_seats_taken INT,
firstclass_seats_taken INT,
CONSTRAINT FLIGHTAVAILABILITY_PK PRIMARY KEY (flight_id,
    segment_number, flight_date),
CONSTRAINT flights_foreign_key2
    FOREIGN KEY(flight_id, segment_number)
    REFERENCES Flights(flight_id, segment_number))

ALTER TABLE CITIES
ADD CONSTRAINT c_uc UNIQUE(city)

**Dependency System**

INSERT and UPDATE statements depend on all constraints on the target table. DELETEs depend on unique, primary key, and foreign key constraints. These statements are invalidated if a constraint is added to or dropped from the target table.
CREATE AGGREGATE statement

The CREATE AGGREGATE statement creates a user-defined aggregate. An aggregate, also known as a set function or a column function, provides a means for evaluating an expression over a set of rows. Cloudscape provides a number of built-in aggregates, such as MAX, MIN, COUNT, and the like. (For more information, see “Aggregates (Set Functions)” on page 1-129.) With CREATE AGGREGATE, you can define your own aggregate functions. For example, whereas MAX gives you the maximum value in a column, you could define an aggregate called MAXBUTONE that would return the second highest value in a column, provided that you created a Java class to implement the aggregation. Another example is standard deviation; with the appropriate Java class to back it up, you could create an aggregate that operates on a set of doubles and returns the standard deviation.

When you create an aggregate, you specify a Java class that implements COM.cloudscape.aggregates.AggregateDefinition.

For information about creating such classes, see “Programming User-Defined Aggregates” on page 5-27 in the Cloudscape Developer’s Guide.

Syntax

CREATE AGGREGATE AggregateName FOR
{ JavaClassName | ClassAlias }

You cannot create an aggregate with the same name as a method alias.

You cannot create an aggregate with the same name as a built-in aggregate.

User-defined aggregates are not permitted in check constraints.

Examples

CREATE AGGREGATE MAXBUTONE FOR
   JBMSTours.aggregates.MaxButOneDef

CREATE AGGREGATE STDEV FOR
   JBMSTours.aggregates.StandardDeviation

SELECT MAX(DISTINCT miles) AS Highest, MAXBUTONE(DISTINCT miles) AS SecondHighest
FROM Flights

SELECT AVG(flying_time), STDEV(flying_time)
FROM Flights
CREATE CLASS ALIAS statement

A CREATE CLASS ALIAS statement allows you to create an alias for a Java class for use within SQL-J statements. Once you create an alias for a class, you no longer have to use the full package name to refer to the class. In addition, when calling static methods, referencing static fields, and casting Java data types, you do not have to use the CLASS keyword when the classes have aliases.

Class aliases are not defined within a particular schema.

Syntax

CREATE CLASS ALIAS [ClassAlias] FOR JavaClassName

Like all SQL92Identifiers, class aliases are case-insensitive unless delimited.

If you do not specify the alias for the class, Cloudscape uses the class name (minus any package names) as the alias. Specifying an alias for the class that is not the actual class name is useful in situations in which two different packages have a class of the same name.

The Java class must be public and available on the current system or database class path.

NEW: Class aliases are new in Version 3.0.

Examples

-- by default, the alias is the class name
-- so in this example, the alias would be TimeZone
CREATE CLASS ALIAS FOR java.util.TimeZone

CREATE CLASS ALIAS tZone FOR
    JBMSTours.serializabletypes.TimeZone

CREATE CLASS ALIAS "timeZone" FOR
    JBMSTours.serializabletypes.TimeZone

-- Once you create an alias, use it where
-- otherwise you would have to use the full
-- class name.
CREATE TABLE tzones (tzone SERIALIZE("timeZone"))
Cloudscape-Supplied Class Aliases for Proprietary Classes

Cloudscape provides class aliases for Cloudscape classes that may be referenced within SQL-J statements. Table 1-1 shows the built-in class aliases.

Table 1-1: Built-in Class Aliases

<table>
<thead>
<tr>
<th>Class Case-Insensitive Alias</th>
<th>Class Case-Sensitive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM.cloudscape.database.ConsistencyChecker</td>
<td>ConsistencyChecker</td>
</tr>
<tr>
<td>COM.cloudscape.database.Database</td>
<td>Database</td>
</tr>
<tr>
<td>COM.cloudscape.database.Factory</td>
<td>Factory</td>
</tr>
<tr>
<td>COM.cloudscape.database.PropertyInfo</td>
<td>PropertyInfo</td>
</tr>
<tr>
<td>COM.cloudscape.database.UserUtility</td>
<td>UserUtility</td>
</tr>
<tr>
<td>COM.cloudscape.synchronization.ErrorTransactionVTI</td>
<td>ErrorTransactionVTI</td>
</tr>
<tr>
<td>COM.cloudscape.synchronization.RowListVTI</td>
<td>RowListVTI</td>
</tr>
<tr>
<td>COM.cloudscape.synchronization.StatementListVTI</td>
<td>StatementListVTI</td>
</tr>
<tr>
<td>COM.cloudscape.synchronization.TransactionListVTI</td>
<td>TransactionListVTI</td>
</tr>
<tr>
<td>COM.cloudscape.tools.dbclasses</td>
<td>dbclasses</td>
</tr>
<tr>
<td>COM.cloudscape.tools.FileExport</td>
<td>FileExport</td>
</tr>
<tr>
<td>COM.cloudscape.tools.FileImport</td>
<td>FileImport</td>
</tr>
<tr>
<td>COM.cloudscape.types.AliasInfo</td>
<td>AliasInfo</td>
</tr>
<tr>
<td>COM.cloudscape.types.DefaultInfo</td>
<td>DefaultInfo</td>
</tr>
<tr>
<td>COM.cloudscape.types.DependableFinder</td>
<td>DependableFinder</td>
</tr>
<tr>
<td>COM.cloudscape.types.ErrorInfo</td>
<td>ErrorInfo</td>
</tr>
<tr>
<td>COM.cloudscape.types.IndexDescriptor</td>
<td>IndexDescriptor</td>
</tr>
<tr>
<td>COM.cloudscape.types.ReferencedColumnsDescriptor</td>
<td>ReferencedColumnsDescriptor</td>
</tr>
<tr>
<td>COM.cloudscape.types.TypeDescriptor</td>
<td>TypeDescriptor</td>
</tr>
<tr>
<td>COM.cloudscape.util.BitUtil</td>
<td>BitUtil</td>
</tr>
<tr>
<td>COM.cloudscape.util.KeyGen</td>
<td>KeyGen</td>
</tr>
<tr>
<td>COM.cloudscape.vti.ExternalQuery</td>
<td>ExternalQuery</td>
</tr>
<tr>
<td>COM.cloudscape.vti.LockTable</td>
<td>LockTable</td>
</tr>
<tr>
<td>COM.cloudscape.vti.TransactionTable</td>
<td>TransactionTable</td>
</tr>
<tr>
<td>COM.cloudscape.vti.TriggerOldTransitionRows</td>
<td>TriggerOldTransitionRows</td>
</tr>
</tbody>
</table>

Class Aliases and Cloudscape synchronization

Except for built-in class aliases, you must explicitly add class aliases to a publication.
CREATE INDEX statement

A CREATE INDEX statement creates an index on a table. Indexes can be on one or more columns in the table.

Syntax

```
CREATE [UNIQUE] [ BTREE] INDEX IndexName
ON TableName ( SimpleColumnName
    [ , SimpleColumnName ] * )
    [ PROPERTIES clause ]
```

A column must not be named more than once in a single CREATE INDEX statement. Different indexes can name the same column.

Cloudscape can use indexes to improve the performance of data manipulation statements (see Tuning Cloudscape). Also, UNIQUE indexes provide a form of data integrity checking.

Index names are unique within a schema. (Some database systems allow different tables in a single schema to have indexes of the same name.)

Both index and table are assumed to be in the same schema if a schema name is specified for one of the names, but not the other. If schema names are specified for both index and table, an exception will be thrown if the schema names are not the same. If no schema name is specified for either table or index, the current schema is used.

For information about how indexes affect query optimization, see Tuning Cloudscape.

A PROPERTIES clause allows you to specify page size for the index and overrides the default page size set on a database-wide basis. See “PROPERTIES clause” on page 1-95.

Indexes cannot be created in the SYS schema.

Indexes and Constraints

Unique, primary key, and foreign key constraints generate indexes that enforce or “back” the constraint (and are thus sometimes called backing indexes). If a column or set of columns has a UNIQUE or PRIMARY KEY constraint on it, you do not need to create an index on those columns. Cloudscape has already created it for you with a system-generated name. System-generated names for indexes that back up constraints are easy to find by querying the system tables if you name your constraint, as recommended by Cloudscape. For example, to find out the name of the index that backs a constraint called FLIGHTS_PK:
### SQL-J Language Reference

```sql
SELECT conglomeratename
FROM sys.sysconglomerates, sys.sysconstraints
WHERE isconstraint=true
AND sys.sysconglomerates.tableid = sys.sysconstraints.tableid
AND constraintname = 'FLIGHTS_PK'
```

See “CONSTRAINT clause” on page 1-29 for more information about constraints.

### Indexes on Java Data Types

You cannot create indexes on Java data type columns unless they are *orderable*. For information about orderable Java data types, see “Orderable Java Data Types” on page 1-214 and “Orderable Java Data Types and Indexes” on page 1-215.

#### Examples

```sql
CREATE BTREE INDEX OrigIndex ON Flights(orig_airport)

CREATE INDEX large_index ON PageContent (large_text_column)
PROPERTIES cloudscape.storage.pageSize=8192

-- create an index on a column
-- that stores an Orderable Java data type
CREATE INDEX ci ON Cities(city)
```

### Page Size and Key Size

**NOTE:** The size of the key columns in an index must be equal to or smaller than half the page size. If the length of the key columns in an index is larger than half the page size of the index, creating an index on those key columns for the table fails. For existing indexes, an insert of new rows for which the key columns are larger than half of the index page size causes the insert to fail.

### Dependency System

Prepared statements that involve SELECT, INSERT, UPDATE, UPDATE WHERE CURRENT, DELETE, and DELETE WHERE CURRENT on the table referenced by the CREATE INDEX statement are invalidated when the index is created. Open cursors on the table are not affected.
CREATE METHOD ALIAS statement

CREATE METHOD ALIAS adds an alias for a static Java method, making it appear to be a native built-in function.

Method aliases are not defined within a particular schema.

Syntax

```
CREATE METHOD ALIAS MethodAlias
FOR { JavaClassName | ClassAlias }.JavaMethodName
```

The Java class must be a public class and available on the current system or database class path. The method must exist and be public. A method alias and a class alias can share the same name.

NEW: Validity checking for method alias creation is new in Version 3.0.

Examples

```
CREATE METHOD ALIAS ABS
FOR java.lang.Math.abs

CREATE METHOD ALIAS findCity
FOR JBMSTours.serializabletypes.City.findCity

-- City is an alias for JBMSTours.serializabletypes.City
CREATE METHOD ALIAS findCity
FOR City.findCity

-- a method alias and class alias can have
-- the same name
CREATE METHOD ALIAS City
FOR City.findCity
```

Once a method alias has been defined within a database, you refer to it with the alias alone:

```
VALUES ABS(-5)

VALUES findCity(getCurrentConnection(), 35)
```
CREATE SCHEMA statement

A schema is a way to logically group objects in a single collection and provide a unique namespace for objects.

Syntax

CREATE SCHEMA SchemaName

The CREATE SCHEMA statement is used to create a schema. Schema names must be unique within the database.

You must create a schema with a CREATE SCHEMA statement before referring to it, unless the schema is the APP or SYS schema.

Examples

-- Create a schema for hotel-related tables
CREATE SCHEMA Hotels

-- Create a schema for airline-related tables
CREATE SCHEMA Flights

-- Create a table called “Availability” in each schema
CREATE TABLE Flights.Availability
  (flight_id CHAR(6),
   segment_number INT,
   flight_date DATE,
   economy_seats_taken INT,
   business_seats_taken INT,
   firstclass_seats_taken INT,
   CONSTRAINT Flight_Availability_PK PRIMARY KEY
    (flight_id, segment_number, flight_date))

CREATE TABLE Hotels.Availability
  (hotel_id INT,
   booking_date DATE,
   rooms_taken INT,
   CONSTRAINT HotelAvailability_PK PRIMARY KEY
    (hotel_id, booking_date))
CREATE STATEMENT statement

The CREATE STATEMENT statement creates stored prepared statements, which are SQL-J statements (usually pre-compiled) stored in the database for multiple use. Storing statements allows you to avoid compilation time and to share compiled statement plans between sessions and connections. A stored prepared statement is prepared, tagged with an identifier, and stored in a system table. The prepared statement can then be invoked directly, using its identifier, without compiling the statement again.

Stored prepared statements provide an additional performance benefit in multi-user systems, because they optimize memory use. See “Storing Prepared Statements to Improve Performance” on page 3-13 in Tuning Cloudscape for more information.

The compiled statement and reference information about the statement are stored in the SYS.SQLSTATEMENTS table, and information about each parameter is stored in the SYS.SYSCOLUMNS table.

Statements cannot be created in the SYS schema.

You cannot create stored prepared statements for CREATE STATEMENT and EXECUTE STATEMENT statements.

Syntax

```sql
CREATE STATEMENT StatementName [ NOCOMPILE ]
AS SQLStatement
[ USING singleRowResultSet ]
```

NEW: Beginning in Version 3.0, you can create a stored prepared statement with the NOCOMPILE keyword, which creates a statement that is not compiled until its first execution or until you explicitly request it.

About the USING Clause

The USING clause, which is optional, allows you to provide sample values for parameters in the CREATE STATEMENT statement by providing a SingleRowResultSet. A SingleRowResultSet is a query that returns a single row, usually a VALUES expression or a SELECT.

This gives the optimizer representative data so that it 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. These values are used for compiling the statement, not for executing the statement. (You provide values for executing the statement when you actually execute it with the EXECUTE STATEMENT statement.)
Providing sample values (with the USING clause) is particularly useful if the application designer has a good idea how the statement will be used in the future.

For example, imagine an indexed column that is used to store ages. Ages typically range from 0 to 100. If your query typically looks for one specific age, the index will prove useful. If your query typically looks for values under a specific value, the index may or may not prove useful. For example, if the query looks for all values less than 5, an index will prove useful, because Cloudscape will avoid scanning 95% of the data pages. However, if the query looks for all values less than 90, it would probably make more sense for Cloudscape to go straight to the data pages and filter out those unneeded rows later. Providing a realistic sample value when creating the statement allows Cloudscape to devise a useful access plan.

**Examples**

```sql
CREATE STATEMENT getFullFlightInfo
AS SELECT flight_id, segment_number, orig_airport, depart_time, dest_airport, arrive_time, flying_time, miles
FROM Flights
WHERE flight_id = ? AND segment_number = ?
USING VALUES ('AA1111', 1)

CREATE STATEMENT getDirectFlights AS SELECT
orig_flights.flight_id
FROM (select flight_id FROM flights WHERE orig_airport = ?) orig_flights,
(select flight_id FROM flights WHERE dest_airport = ?) dest_flights
WHERE orig_flights.flight_id = dest_flights.flight_id
USING VALUES ('SFO', 'SCL')

CREATE STATEMENT INSERTFlightObject
AS INSERT INTO FlightObjects VALUES (?, ?)

CREATE STATEMENT UpdateCursor NOCOMPILE AS
UPDATE FlightAvailability SET economyseats_taken =
    ((economyseats_taken IS NULL) ? 0 : economyseats_taken) + ?
WHERE CURRENT OF FAROWS
```

**Dependency System**

A stored prepared statement depends on all the objects that are referenced by the statement. A stored prepared statement can be dependent on the following types of objects:
• cursors (for positioned update or delete)
• tables
• views
• indexes
• constraints
• publications
• aliases (method aliases, class aliases, aggregates, work units)
• triggers

If an object is referenced by a stored prepared statement, the object cannot be
dropped or altered in such a way that the statement is invalidated if the prepared
statement is currently in use. For example, if a stored prepared statement uses the
orig_index on the Flights table to access its data, that index cannot be dropped when
that stored prepared statement is in use (that is, if a ResultSet is open on the
statement). This is true of any active prepared statement (stored or not).

If the statement is not in use, a referenced object can be dropped and the stored
prepared statement is invalidated. The invalidation is recorded by setting the
VALID field of SYS.SYSSTATEMENTS to FALSE. The next time the statement is
executed, it is recompiled before execution. If an object on which it depends does
not exist at the time it is recompiled, the execution request results in an error. If a
new object is created with the same name, the statement will be compiled against
the new object. In the example in the preceding paragraph, if a new index on the
Flights table is created, the stored prepared statement will be compiled with that
new index being considered as a possible access path. This is the same behavior as
that of a prepared statement that is not stored in Cloudscape. You can also force
recompilation. See “ALTER STATEMENT statement” on page 1-18.

Publications depend on stored prepared statements if the statements are part of the
publications. Adding or dropping publications may invalidate stored prepared
statements.
CREATE TABLE statement

A CREATE TABLE statement creates a table. Tables contain columns and constraints that specify a column or columns. Columns have a data type and can specify column constraints. (A constraint is a rule to which data must conform.) For information about constraints, see “CONSTRAINT clause” on page 1-29.

You can specify a default value for a column. A default value is the value to be inserted into a column if no other value is specified. If not explicitly specified, the default value of a column is NULL.

**NEW:** The ability to specify a default for a column is new in Version 3.0.

The SET LOCKING clause allows you to override row-level locking for the specific table, if your system is set for row-level locking. (If your system is set for table-level locking, you cannot change the locking granularity to row-level locking.) To override row-level locking for the specific table, set locking for the table to TABLE. After you create such a table, you can change locking back to ROW with the a SET LOCKING clause in the ALTER TABLE statement.

**NEW:** The SET LOCKING clause is new in Version 3.0.

You can specify storage properties such as page size for a table with a PROPERTIES clause.

Tables cannot be created in the SYS schema.

**Syntax**

```
CREATE TABLE TableName 
( { ColumnDefinition | Table-Level Constraint }
[ , { ColumnDefinition | Table-Level Constraint } ] * )
[ PROPERTIES clause ]
[ SET LOCKING = { TABLE | ROW } ]
```

**ColumnDefinition**

```
SimpleColumnName DataType
[ [ WITH ] DEFAULT { ConstantExpression | NULL }]
[ Column-Level Constraint ]*
```

The syntax of `DataType` is described in “Data Types” on page 1-180.

The syntaxes of `Column-Level Constraint` and `Table-Level Constraint` are described in “CONSTRAINT clause” on page 1-29.
Column Defaults

For the definition of a default value, a `ConstantExpression` is an expression that does not refer to any table. It can include literals, built-in functions such as `USER`, `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP`, Java method invocations, Java field references, and any other expression, as long as it does not refer directly to a table or to a column in a table. (If the default table contains a method call, the method may in fact contain SQL-J statements that refer to tables and columns.) A method call cannot contain dynamic parameters (\?).

In a Cloudscape synchronization system, non-literal defaults are re-evaluated at the source if the insert or updated statement is executed within a work unit.

Examples

```sql
CREATE TABLE Cities (
    city_id INT CONSTRAINT CITIES_PK PRIMARY KEY,
    city SERIALIZE(City),
    country_ISO_code CHAR(2)
) CONSTRAINT countries_fk REFERENCES Countries);
```

```sql
CREATE TABLE HotelAvailability
(hotel_id INT,
booking_date DATE,
rooms_taken INT DEFAULT 0,
PRIMARY KEY (hotel_id, booking_date))
```

```sql
CREATE TABLE CustomizedTours(
    group_id INT PRIMARY KEY,
    customized_tour SERIALIZE(Tour))
```

**NOTE:** For more examples of CREATE TABLE statements using the various constraints, see “CONSTRAINT clause” on page 1-29.
Interaction with Java Data Types

You can create columns that store Java objects. For more information, see “Java Data Types (User-Defined Data Types)” on page 1-211.

You can store SQL NULL values in Java data type columns (assuming that the columns do not have NOT NULL constraints). (An SQL NULL value is not the same thing as a Java null reference, although they do map to one another.)

Published Tables

When a table is published, it has the same lock granularity at the target as at the source. The SET LOCKING clause of the CREATE TABLE and ALTER TABLE statements is implicitly published to the targets whose publication includes the affected table.

Column defaults are always implicitly published if a column is published. A column in a target cannot have a different default value than the corresponding column in the source.
CREATE TRIGGER statement

A trigger defines a set of actions that are executed when a database event occurs on a specified table. A database event is a delete, insert, or update operation. For example, if you define a trigger for a delete on a particular table, the trigger’s action occurs whenever someone deletes a row or rows from the table.

Along with constraints, triggers can help enforce data integrity rules with actions such as cascading deletes or updates. Triggers can also perform a variety of functions such as issuing alerts, updating other tables, sending e-mail, and other useful actions.

You can define any number of triggers for a single table, including multiple triggers on the same table for the same event.

You can create a trigger in any schema except SYS. The trigger need not reside in the same schema as the table on which it is defined.

You cannot create triggers on any table in the SYS schema.

Syntax

CREATE TRIGGER TriggerName
{ BEFORE | AFTER }
{ INSERT | DELETE | UPDATE [ OF ColumnName [, ColumnName]* ]
ON TableName
[ ReferencingClause ]
[ FOR EACH { ROW | STATEMENT } ]
TriggerAction

ReferencingClause

REFERENCING
{
  ( OLD | NEW ) [ ROW ] [ AS ] CorrelationName |
  ( OLD | NEW ) TABLE [ AS ] Identifier
}

Before or After: When Triggers Fire

Triggers are either before or after triggers (in Syntax above, see the second line):

- *Before* triggers fire before the statement’s changes are applied and before any constraints have been applied. Before triggers can be either row or statement triggers (see “Statement vs. Row Triggers” on page 1-53).
- *After* triggers fire after all constraints have been satisfied and after the changes have been applied to the target table. After triggers can be either row or statement triggers (see “Statement vs. Row Triggers” on page 1-53).
Insert, Delete, or Update: What Causes the Trigger to Fire

A trigger is fired by one of the following database events, depending on how you define it (in Syntax above, see the third line):

- INSERT
- UPDATE
- DELETE

You can define any number of triggers for a given event on a given table. For update, you can specify columns.

Referencing Old and New Values: The Referencing Clause

Many trigger actions need to refer to data that is currently being changed by the database event that caused them to fire. The trigger action may need to refer to the old (pre-change, or “before”) values or the new (post-change or “after”) values.

Cloudscape provides you with a number of ways to refer to data that is currently being changed by the database event that caused the trigger to fire. The easiest way to refer to the changed data in the trigger action is use the transition variables or transition tables.

For row triggers, the transition variables NEW and OLD refer to the after and before image of the changes made to a row being modified by the current event. For example, you can refer to the transition variable “OLD” in the trigger action:

DELET E FROM HotelAvailability WHERE hotel_id = OLD.hotel_id

The referencing clause allows you to provide a correlation name or alias for these transition variables by specifying OLD/NEW AS correlationName if you do not wish to use the default variable names.

For example, if you add the following clause to the trigger definition (the word “row” is optional):

REFERENCING OLD ROW AS DELETEDROW

you can then refer to this correlation name in the trigger action:

DELETE FROM HotelAvailability WHERE hotel_id = DELETEDROW.hotel_id

The OLD and NEW transition variables map to a java.sql.ResultSet with a single row.
**NOTE:** Only row triggers (see “Statement vs. Row Triggers” on page 1-53) can use the transition variables. INSERT row triggers cannot reference an OLD row. DELETE row triggers cannot reference a NEW row.

For statement triggers, transition *tables* serve as a table identifier for the trigger action or the trigger qualification. The default identifier for transition tables are NEW and OLD for the after and before row sets, respectively; you must specify the word TABLE after the keyword NEW or OLD.

For example, you can refer to the transition table “OLD” in the trigger action:

```
DELETE FROM HotelAvailability WHERE hotel_id IN (SELECT hotel_id FROM OLD)
```

You can supply an alternate identifier to be used in place of the default identifiers by adding a referencing clause that specifies OLD/NEW TABLE AS *identifier*. For example:

```
REFERENCING OLD TABLE AS DeletedHotels
```

allows you to use that new identifier (*DeletedHotels*) in the trigger action:

```
DELETE FROM HotelAvailability WHERE hotel_id IN (SELECT hotel_id FROM DeletedHotels)
```

The old and new transition tables map to a `java.sql.ResultSet` with cardinality equivalent to the number of rows affected by the triggering event.

**NOTE:** Only statement triggers (see “Statement vs. Row Triggers” on page 1-53) can use the transition tables. INSERT statement triggers cannot reference an OLD table. DELETE statement triggers cannot reference a NEW table.

The referencing clause can designate only one new correlation or identifier and only one old correlation or identifier. Row triggers cannot designate an identifier for a transition table and statement triggers cannot designate a correlation for transition variables.

**NOTE:** There are other ways to reference old and new values; see “Trigger Execution Context” on page 1-55 and “Transition Row VTIs” on page 1-56.

**Statement vs. Row Triggers**

By default, a trigger is a *statement trigger* unless you specify that it is a *row trigger* (in Syntax above, see the sixth line):
• **statement triggers**
  A statement trigger fires once per triggering event and regardless of whether any rows are modified by the insert, update, or delete event.

• **row triggers**
  A row trigger fires once for each row affected by the triggering event. If no rows are affected, the trigger does not fire.

**NOTE:** An update that sets a column value to the value that it originally contained (for example, `UPDATE T SET C = C`) causes a row trigger to fire, even though the value of the column is the same as it was prior to the triggering event.

**TriggerAction**

The action defined by the trigger is called the trigger action (in Syntax above, see the last line). It can be any valid SQL-J statement, including one with Java method calls. It has the following limitations:

• It must not contain any dynamic parameters (?).
• It must not create, alter, or drop the table upon which the trigger is defined.
• It must not add an index to or remove an index from the table on which the trigger is defined.
• It must not add a trigger to or drop a trigger from the table upon which the trigger is defined.
• It must not commit or roll back the current transaction or change the isolation level.
• It must not execute a stored prepared statement with the USING clause.
• If it is a before trigger, it must not perform an insert, update, or delete on the table on which it was defined.
• It does not issue a SET CONSTRAINTS statement or SET TRIGGERS statement on the table on which the trigger is defined.

For more information on trigger actions, see “Programming Trigger Actions” on page 5-30 in the *Cloudscape Developer’s Guide*.

**Order of Execution**

When a database event occurs that fires a trigger, Cloudscape performs actions in this order:

• It fires *before* triggers.
CREATE TRIGGER statement

- It performs constraint checking (primary key, unique key, foreign key, check).
- It performs the insert, update, or delete.
- It fires *after* triggers.

When multiple triggers are defined for the same database event for the same table for the same trigger time (before or after), triggers are fired in the order in which they were created.

**Trigger Execution Context**

You can access information about a trigger’s context from a new structure, `COM.cloudscape.database.TriggerExecutionContext`. You obtain it by calling `COM.cloudscape.database.Factory.getTriggerExecutionContext()` from within a TriggerAction (or from within a Java method called by a TriggerAction). For example, within a Java method (executed by a TriggerAction):

```java
TriggerExecutionContext context =
    COM.cloudscape.database.Factory.getTriggerExecutionContext();
ResultSet oldrow = context.getOldRow();
```

Within an SQL statement:

```sql
CREATE TRIGGER . . .
DELETE FROM Hotels WHERE city_id =
    Factory.getTriggerExecutionContext().
    getOldRow().getInt("CITY_ID");
```

In addition to access to before and after rows, the `TriggerExecutionContext` provides the following information:

- The columns changed by the triggering event.
- The type of event that fired the trigger.
- The SQL text of the triggering database event (for example, the text of the INSERT statement that caused the trigger to fire). (When this trigger is fired by the replay of a statement during a refresh, the statement text is null.)
- The “before” image of the row being changed.
- The “after” image of the row being changed.
- The target table unique identifier.
- The name of the target table.

For complete information on how to use this object, see the Javadoc for `COM.cloudscape.database.TriggerExecutionContext`.  

Cloudscape Version 3.0
Transition Row VTIs

The class `COM.cloudscape.vti.TriggerOldTransitionRows` is a VTI wrapper around a `TriggerExecutionContext.getOldRowSet()`. Selecting an instantiation of this VTI is the equivalent of the OLD TABLE transition table. The only difference is that this VTI is accessible from within an SQL-J statement executed by a Java method called by the trigger action; the OLD transition table is accessible only by the trigger action’s SQL.

For example, the following two trigger actions are identical:

```sql
DELETE FROM HotelAvailability WHERE hotel_id IN
(SELECT hotel_id FROM OLD)
DELETE FROM HotelAvailability WHERE hotel_id IN
(SELECT hotel_id FROM NEW TriggerOldTransitionRows() AS
DeletedHotels)
```

You may want to call a Java method in the trigger action to perform some set of actions. In the body of this method, you would not be able to use the OLD or NEW transition tables, so you would have to use the VTI instead.

For example:

```sql
-- TriggerActions is a class alias
CREATE TRIGGER ShowNewBookings
AFTER INSERT
ON HotelBookings
FOR EACH STATEMENT
CALL TriggerActions.showNewBookings(getCurrentConnection())
```

```java
public static void showNewBookings(Connection conn) throws SQLException {
    Statement s = conn.createStatement();
    Util.println("New hotel bookings:");
    ResultSet rs = s.executeQuery("SELECT * FROM new TriggerNewTransitionRows() AS EQ");
    while (rs.next()) {
        Util.println(rs.getInt(1) + " " +
                     rs.getDate(3) + " " + rs.getDate(4));
    }
    rs.close();
    s.close();
}
```
The class `COM.cloudscape.vti.TriggerNewTransitionRows` is a VTI wrapper around a `TriggerExecutionContext.getNewRowSet()`. Selecting from this VTI is the equivalent of the NEW TABLE transition table. The only difference is that this VTI is accessible from within an SQL-J statement executed by a Java method called by the trigger action; the NEW transition table is accessible only by the trigger action’s SQL.

For examples of using the VTIs within Java methods called by the trigger action, see “Programming Trigger Actions” on page 5-30 in the Cloudscape Developer’s Guide.

**Examples**

```sql
-- A statement and before trigger,
-- uses a REFERENCING clause to give an alternate
-- name to a transition table.
-- When a hotel is deleted, delete related rows in
-- the HOtelAvailability table.
CREATE TRIGGER HotelsCascadingDelete
BEFORE DELETE
ON Hotels
REFERENCING OLD Table AS DeletedHotels
FOR EACH STATEMENT
DELETE FROM HotelAvailability WHERE hotel_id IN
(SELECT hotel_id FROM DeletedHotels)

-- a statement and before trigger that uses special VTI
-- Before deleting a row in the hotels table,
-- delete related rows in the HotelAvailability table,
-- which are in a foreign key relationship with the Hotels
-- table
CREATE TRIGGER HotelsCascadingDelete1
BEFORE DELETE
ON Hotels
FOR EACH STATEMENT
DELETE FROM HotelAvailability WHERE hotel_id IN
(SELECT hotel_id FROM NEW TriggerOldTransitionRows() AS
DeletedHotels)

-- use a transition variable in a row trigger
CREATE TRIGGER HotelsCascadingDelete3
BEFORE DELETE
ON Hotels
FOR EACH ROW
DELETE FROM HotelAvailability WHERE hotel_id = OLD.hotel_id
```

*Cloudscape Version 3.0*
-- For each row being deleted,
-- call a Java method. This method deletes related
-- rows in the HotelBookings, unless the bookings are
-- for a future date. In that case, it changes the booking
-- to a new hotel (one that isn't being deleted)
CREATE TRIGGER HotelsCascadingDelete2
BEFORE DELETE
ON Hotels
FOR EACH ROW
CALL TriggerActions.deleteOrRedoHotelBooking(
    getCurrentConnection(), CURRENT_JDATE())

-- automatically update the column that records
-- the return value of a method call
CREATE TRIGGER pseudoMethodIndexInsert
AFTER
INSERT
ON CustomizedTours
FOR EACH STATEMENT
UPDATE CustomizedTours SET getTotalCost =
    customized_tour.getTotalCost()
WHERE group_id IN (SELECT group_id FROM new
    TriggerNewTransitionRows() AS NewRows)

-- perform some actions if only specific columns
-- are modified
CREATE TRIGGER hotelsAltered
AFTER UPDATE OF normal_rate, high_season_rate
ON Hotels
FOR EACH STATEMENT
    CALL TriggerActions.hotelChanged(getCurrentConnection())

NOTE: You can find more examples in “Programming Trigger Actions” on page 5-30 in the Cloudscape Developer’s Guide.

Trigger Recursion
It is possible for one trigger to cause another trigger to fire, and thus it is possible
for triggers to recurse infinitely. Cloudscape supplies the property
cloudscape.language.triggerMaximumRecursionLevel, which specifies the
maximum recursion level, to allow you to limit trigger recursion. The default value
of this property is 16. For more information, see
“cloudscape.language.triggerMaximumRecursionLevel” on page 5-43 in Tuning Cloudscape.
Special Functions and Triggers

Special system functions that return information about the current time or current user are evaluated when the trigger fires, not when it is created. Such functions include:

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- SESSION_USER
- USER

Cloudscape synchronization and Triggers

In a Cloudscape synchronization triggers are not published by default. For information, see the Cloudscape Synchronization Guide.

Dependency System

The trigger action may reference database objects other than the table upon which the trigger is declared. If any of these database objects is dropped, the trigger is invalidated. If the trigger cannot be successfully recompiled upon the next execution, the invocation throws an exception and the statement that caused it to fire will be rolled back.
CREATE VIEW statement

Views are virtual tables formed by a query. A view is a dictionary object that you can use until you drop it.

Views are not updatable.

Views cannot be created in the SYS schema.

Syntax

CREATE VIEW ViewName
    [ ( SimpleColumn|Name [, SimpleColumn|Name] * ) ]
AS Query

A view definition can contain an optional view column list to explicitly name the columns in the view. If there is no column list, the view inherits the column names from the underlying query, including any internally generated names for unnamed columns. All columns in a view must be uniquely named.

Example

CREATE VIEW v1 (col_sum, col_diff)
AS SELECT c1 + c2, c1 - c2
FROM Table1

CREATE VIEW BigStates (StateName)
AS VALUES 'Alaska', 'California', 'Texas'

CREATE VIEW Segments_SeatBookings
    (flight_id, segment_number, number_seats, travel_date)
AS SELECT flight_id, segment_number, number_seats, travel_date
FROM FlightBookings
UNION ALL
SELECT segment_two_flight_id AS flight_id,
    segment_two_segment_number AS segment_number,
    number_seats, travel_date
FROM FlightBookings
WHERE segment_two_flight_id IS NOT NULL

Dependency System

View definitions are dependent on the tables, views, and aliases referenced within the view definition. DML (data manipulation language) statements that contain view references depend on those views, as well as the objects in the view definitions.
that the views are dependent on. In Version 3.0, a view is not dependent on indexes. Statements that reference the view depend on indexes the view uses; which index a view uses may change from statement to statement based on how the query is optimized. For example, given:

```sql
CREATE TABLE T1 (C1 DOUBLE PRECISION)
CREATE METHOD ALIAS sin
  FOR java.lang.Math.sin
CREATE VIEW V1
  AS
  SELECT sin(C1)
  FROM T1
```

the following SELECT:

```sql
SELECT *
FROM V1
```

is dependent on view V1, table T1, and method alias sin.
CROSS JOIN

A CROSS JOIN is one of the JOIN operations. It provides a way to specify a Cartesian product between two tables: it does not allow you to specify the join column. You can, however, specify a WHERE clause.

Syntax

TableExpression CROSS JOIN TableExpression

Example

SELECT *
FROM Flights CROSS JOIN Airlines
WHERE orig_airport = 'SFO'
AND dest_airport = 'LAX'

SELECT *
FROM (VALUES (1, 2), (3, 4)) AS Peter(apples, oranges)
CROSS JOIN
(VVALUES ('a', 'b'), ('c', 'd')) AS Paul (bananas, pineapples)

<table>
<thead>
<tr>
<th>APPLES</th>
<th>ORANGES</th>
<th>BANANAS</th>
<th>PINEAPPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
DELETE statement

Syntax

{  
    DELETE FROM TableName [WHERE clause] |  
    DELETE [FROM TableName] WHERE CURRENT OF clause  
}

The first syntactical form, called a searched delete, removes all rows identified by the table name and WHERE clause.

The second syntactical form, called a positioned delete, deletes the current row of an open, updatable cursor. If there is no current row or if it no longer satisfies the cursor’s query, an exception is raised. For more information about updatable cursors, see “SELECT statement” on page 1-106.

Example

DELETE FROM HotelBookings  
WHERE arrival < CURRENT_DATE

stmt.executeUpdate("DELETE WHERE CURRENT OF " +  
    resultSet.getCursorName());

Dependency System

A searched delete statement depends on the table being updated, all of its conglomerates (units of storage such as heaps or indexes), and any other table named in the WHERE clause. A CREATE or DROP INDEX statement for the target table of a prepared searched delete statement invalidates the prepared searched delete statement.

The positioned delete statement depends on the cursor and any tables the cursor references. You can compile a positioned delete even if the cursor hasn’t been opened yet. However, removing the open cursor with the JDBC close method invalidates the positioned delete.

A CREATE or DROP INDEX statement for the target table of a prepared positioned delete invalidates the prepared positioned delete statement.

Searched delete statements depend on all aliases used in a statement. Dropping an alias invalidates a prepared delete statement if the statement uses alias.
DROP AGGREGATE statement

DROP AGGREGATE drops a user-defined aggregate.

Syntax

```
DROP AGGREGATE AggregateName
```

Dependency System

If a view references an aggregate, you cannot drop an aggregate. Dropping an aggregate invalidates statements that depend on the aggregate.
DROP CLASS ALIAS statement

DROP CLASS ALIAS removes the specified class alias. You cannot drop a built-in class alias.

Syntax

DROP CLASS ALIAS ClassAlias

Example

DROP CLASS ALIAS Person

Dependency System

A class alias cannot be dropped if there is an object dependent on it such as a publication, view, constraint, or open statement. Tables are not dependent upon class aliases; a class alias is simply a syntactic shortcut for providing the type definition for columns and parameters.

If there is an open cursor or a view that uses the class alias being dropped, the DROP CLASS ALIAS statement generates an error and does not drop the method alias. Dropping a class alias invalidates other (closed) statements that depend on the class alias.
DROP INDEX statement

DROP INDEX removes the specified index.

Syntax

```
DROP INDEX IndexName
```

Example

```
DROP INDEX OrigIndex
DROP INDEX DestIndex
```

Dependency System

If there is an open cursor on the table from which the index is dropped, the DROP INDEX statement generates an error and does not drop the index. Otherwise, statements that depend on the index’s table are invalidated.
DROP METHOD ALIAS statement

DROP METHOD ALIAS removes the specified method alias.

Syntax

DROP METHOD ALIAS MethodAlias

Example

DROP METHOD ALIAS ABS

Dependency System

A method alias cannot be dropped if there is an object dependent on it such as a publication, view, constraint, or open statement.

If there is an open cursor or a view that uses the method alias being dropped, the DROP METHOD ALIAS statement generates an error and does not drop the method alias. Dropping a method alias invalidates other (closed) statements that depend on the method alias.
DROP SCHEMA statement

The DROP SCHEMA statement drops a schema. The target schema must be empty for the drop to succeed.

Neither the APP schema (the default user schema) nor the SYS schema can be dropped. If the current schema is dropped and it is the user’s default schema, the user’s default schema is set to APP.

Syntax

```
DROP SCHEMA SchemaName
```

Example

```
-- Drop the hotel schema
DROP SCHEMA hotel
```
The DROP STATEMENT statement drops a stored prepared statement created with CREATE STATEMENT. A statement cannot be dropped if it is currently in use. Statements in the SYS schema cannot be dropped by users.

**Syntax**

```
DROP STATEMENT StatementName
```

**Example**

```
-- Drop a statement
DROP STATEMENT insertFlightObject
```
DROP TABLE statement

DROP TABLE removes the specified table.

Syntax

DROP TABLE TableName

Example

DROP TABLE Hotels

Dependency System

If there is an open cursor or a view that uses the table being dropped, then the DROP TABLE statement generates an error and does not drop the table. Dropping a table invalidates other statements that depend on the table.

When a table is dropped, all triggers on that table are automatically dropped. (You don’t have to drop a table’s triggers before dropping the table.)
**DROP TRIGGER statement**

DROP TRIGGER removes the specified trigger.

**Syntax**

```
DROP TRIGGER TriggerName
```

**Example**

```
DROP TRIGGER HotelsCascadingDelete1
```

**Dependency System**

When a table is dropped, all triggers on that table are automatically dropped. (You don’t have to drop a table’s triggers before dropping the table.)
**DROP VIEW statement**

Drops the specified view.

**Syntax**

```
DROP VIEW ViewName
```

**Example**

```
DROP VIEW Segments_SeatBookings
```

**Dependency System**

Any statements referencing the view are invalidated on a DROP VIEW statement. DROP VIEW is disallowed if there are any views or open cursors dependent on the view. The view must be dropped before any objects that it is dependent on can be dropped.
EXECUTE STATEMENT statement

Executes stored prepared statements created with CREATE STATEMENT. EXECUTE STATEMENT locates the specified stored prepared statement and executes it (without recompiling if possible). The statement returns the same ResultSet or update counts that it would if it were prepared directly. If the statement has been invalidated, it is automatically recompiled upon execution, and the new execution information is updated in the relevant system tables.

Syntax

EXECUTE STATEMENT StatementName
[ USING SingleRowResultSet ]

A SingleRowResultSet is a query that returns a single row, usually a VALUES expression or a SELECT. If used, the SingleRowResultSet supplies the parameters for the execution of the statement. See “Supplying Parameters” on page 1-74.

NOTE: Avoid using stored prepared statements for positioned updates and deletes. They are useful only if the cursor that the stored prepared statement uses to position itself exists at the time the stored prepared statement is created and will exist at the time the stored prepared statement is executed, a situation that will probably never occur!

Working with EXECUTE STATEMENT in JDBC

Within a Java program, you can execute a stored prepared statement using the EXECUTE STATEMENT statement in one of the following ways:

• preparing the statement as a PreparedStatement, then executing it one or more times

When an EXECUTE STATEMENT is prepared, the stored prepared statement is retrieved. The prepared statement can then be executed as if it had been prepared “from scratch.” If the statement is to be executed many times by the application, prepare the EXECUTE STATEMENT statement only once to avoid repeated Cloudscape system table scans to retrieve a particular stored prepared statement.

• executing the statement once as a java.sql.Statement

If you need to supply parameters, you will have to use the USING clause with a SingleRowResultSet (see “Supplying Parameters” on page 1-74). This is useful within ij or when testing your query. Performance-wise, it is much better to prepare the statement and supply parameters with JDBC (see previous bulleted item).
Supplying Parameters

There are two different ways to pass parameters to an EXECUTE STATEMENT statement:

- explicitly setting each value using the JDBC setXXX methods in java.sql.PreparedStatement
  To explicitly set the parameter values, create a prepared statement for the EXECUTE STATEMENT statement. Then set the parameters by position using the setXXX methods on the prepared statement. Execute the statement using the execute method on the prepared EXECUTE STATEMENT statement.

- using the USING clause with a SingleRowResultSet
  To supply a SingleRowResultSet set as an argument to the EXECUTE STATEMENT statement, use any statement that returns a single row. The elements in the SingleRowResultSet are mapped directly to the parameters in the statement by position. For example, if there are two parameters for a given statement and a VALUES clause with two elements, the first parameter is set to the first element in the VALUES clause, and so on.
  If the argument to an EXECUTE STATEMENT evaluates to a multiple row ResultSet, an exception is raised.

**NOTE:** The SQL-J statement that provides the SingleRowResultSet has its own compilation and execution costs, so it is more costly than explicitly setting parameter values.

Examples

```
// Execute a prepared statement and supply paramters via JDBC
PreparedStatement getDirectFlights = conn.prepareStatement("EXECUTE STATEMENT GETDIRECTFLIGHTS");
// getDirectFlights now points to the GETDIRECTFLIGHTS statement
// Set the parameters using the setXXX() methods
getDirectFlights.setString(1, 'SFO');
geDirectFlights.setString(2, 'JFK');
ResultSet rs = getDirectFlights.executeQuery();
while(rs.next()) {
    /* retrieve data, etc. */
}
// try Miami, too
getDirectFlights.setString(2, 'MIA');
ResultSet rs = getDirectFlights.executeQuery();
```
-- Execute a prepared statement using parameters passed
-- in with a VALUES clause
EXECUTE STATEMENT getFullFlightInfo
USING VALUES ('AA1111', 1);

-- Execute a prepared statement using parameters from
-- a SELECT statement. Explicitly specify the schema.
EXECUTE STATEMENT APP.getDirectFlights
USING SELECT customized_tour.getCity1(
    getCurrentConnection()).getAirport(),
    customized_tour.getCity2(
        getCurrentConnection()).getAirport()
FROM CUSTOMIZEDTOURS WHERE group_id = 1
Cloudscape’s Virtual Table Interface (VTI) allows an object that implements the `java.sql.ResultSet` interface (and meets other minimal requirements) to appear as an `ExternalVirtualTable`, a legal form of a `TableExpression` in the `FROM` clause of a `SELECT` statement. As far as the rest of the query is concerned, there is no difference between an `ExternalVirtualTable` and a table, view, or other virtual table such as a subquery or `VALUES` clause. The `SELECT` statement can reference any column from an `ExternalVirtualTable` in any of its clauses.

The use of a VTI in SQL-J, similar to the way Java objects can be referenced in other constructs, is dynamic, not declarative; there is no `CREATE EXTERNAL VIRTUAL TABLE` statement. Instead, you call the constructor of the class that implements the VTI requirements, to instantiate an `ExternalVirtualTable` on the fly within the statement.

The VTI is a powerful tool that can be used to make any external data source, such as a flat text file, a live data feed, or a gateway to an operational RDBMS, appear as a table to Cloudscape.

**NOTE:** For information on the requirements for a VTI class and how to develop one, see “Programming VTIs” on page 5-16 in the Cloudscape Developer’s Guide.

**Syntax**

```
NEW { JavaClassName | ClassAlias } (parameterList)
[AS] CorrelationName
[ (SimpleColumnName [ , SimpleColumnName]* ) ]
```

As with other derived tables, you must give an `ExternalVirtualTable` a `CorrelationName`. Names of derived columns are optional.

**NOTE:** Column names returned by a VTI are treated as case-sensitive, delimited identifiers. Provide names of derived columns in the correlation name to treat them as case-insensitive names, or program your VTI to return all uppercase column names.

**Examples**

```
-- FileImport is an alias for the class
-- COM.cloudscape.tools.FileImport is a class
-- provided by Cloudscape that fulfills the VTI requirements.
-- You use the NEW keyword to use it to create an
-- ExternalVirtualTable from data in a flat file.
INSERT INTO WorldCupStatistics
```

*Cloudscape Reference Manual*
SELECT * FROM NEW FileImport('c:/cloudscape/demo/programs/tours/scripts/wc_stat.dat') AS myExternalData

-- jarvti is a class alias for a sample VTI class
-- (JBMSTours.vti.jdbc1_2.jarvti)
-- count the number of classes in a jar file
SELECT COUNT(*)
FROM
NEW jarvti('c:/cloudscape/lib/cloudscape.jar') AS EQ

**Dependency System**

A statement is not dependent on a VTI. The statement that references the VTI is not dependent on any of the objects referenced within the VTI. (As in a java method call, any statements nested within the VTI are dependent on the objects that they reference.)

**Inserting from Table in Same Database**

Be careful when using an `ExternalVirtualTable` to reference a table in the current database. If the `ExternalVirtualTable` is accessing a table in the current database, and you are using that `ExternalVirtualTable` as the source of an INSERT, the query could result in an infinite loop. Cloudscape does not detect that situation. For example, the following statement would result in an infinite loop:

-- INSERT INTO FlightAvailability
-- SELECT * FROM NEW ExternalQuery(
--   'jdbc:cloudscape:toursDB',
--   'select * FROM FlightAvailability') AS eq

**NEW:** Beginning in Version 3.0, the optimizer chooses the correct join order for statements involving a join with an `ExternalVirtualTable`. You no longer have to specify the join order.

If the statement involves two `ExternalVirtualTables`, each of which references the other, the statement cannot be evaluated and is not permitted, as in the following example:

SELECT *
FROM NEW v1(v2.c) AS v1, NEW v2(v1.c) AS v2

**NOTE:** If a VTI cannot be instantiated more than once and it takes a join column as a parameter, no legal join order is possible, and an exception is thrown.
FOR UPDATE clause

The FOR UPDATE clause is an optional part of a SELECT statement. The FOR UPDATE clause specifies whether the ResultSet of a simple SELECT statement that meets the requirements for a cursor is updatable or not. For more information about updatability, see “Requirements for Updatable Cursors” on page 1-107.

Syntax

FOR
{
  READ ONLY | UPDATE [ OF SimpleColumnName [ , SimpleColumnName]* ]
}

SimpleColumnName refers to the names visible for the table specified in the FROM clause of the underlying query.

Cursors are read-only by default. For a cursor to be updatable, you must specify FOR UPDATE.

To restrict the columns that are updatable through the cursor, you can specify a list of column names. 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. For more information about how indexes affect cursors, see Tuning Cloudscape.

Example

SELECT hotel_id, booking_date, rooms_taken
FROM HotelAvailability
FOR UPDATE OF rooms_taken
**FROM clause**

The FROM clause is a mandatory clause in a *SelectExpression*. It specifies the real or virtual tables from which the other clauses of the query can access columns for use in expressions.

**Syntax**

```
FROM [ PROPERTIES joinOrder = { FIXED | UNFIXED } ]
  TableExpression [ , TableExpression ]*
```

The optional PROPERTIES clause before the *TableExpressions* 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 more information, see “joinOrder” on page 6-9 in *Tuning Cloudscape*.

**Examples**

```
SELECT Cities.city_id, Cities.city.showTemperature()
FROM Cities
WHERE city_id < 5

-- other types of TableExpressions
SELECT *
FROM sys.sysconglomerates AS C JOIN sys.systables AS T
  USING (tableid)
WHERE T.tablename = 'CITIES'

-- force the join order
SELECT *
FROM Flights, FlightAvailability
WHERE FlightAvailability.flight_id = Flights.flight_id
  AND FlightAvailability.segment_number = Flights.segment_number
  AND Flights.flight_id < 'AA1115'

-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
SELECT country, city.getName(), hotel_name, normal_rate
FROM COUNTRIES LEFT OUTER JOIN Cities
  ON Countries.country_ISO_code=Cities.country_ISO_code
  LEFT OUTER JOIN Hotels
  ON Cities.city_id=Hotels.city_id
```
GROUP BY clause

A GROUP BY clause, part of a SelectExpression, groups a result into subsets that have matching values for one or more columns. In each group, no two rows have the same value for the grouping column or columns. NULLs are considered equivalent for grouping purposes.

You typically use a GROUP BY clause in conjunction with an aggregate expression.

Syntax

GROUP BY ColumnName [ , ColumnName ] *

ColumnName must be a column from the current scope of the query; there can be no columns from a query block outside the current scope. For example, if a GROUP BY clause is in a subquery, it cannot refer to columns in the outer query.

SelectItems in the SelectExpression with a GROUP BY clause must contain only aggregates or grouping columns.

ColumnName cannot be an expression (although the column name can be qualified by a correlation name given to the table), as per the SQL-92 standard. This makes it difficult to group by method invocations, which are expressions. To work around this, enclose the method invocation in a subquery. For example:

```sql
SELECT MAX(city_temperature), country_name
FROM
  (SELECT city.showTemperature(), city.getDisplayCountry()
   FROM Cities)
AS CT(city_temperature, country_name)
GROUP BY CT.country_name
```

Examples

-- find the average flying_time of flights grouped by
-- airport
SELECT AVG (flying_time), orig_airport
FROM Flights
GROUP BY orig_airport

SELECT MAX(city), region
FROM Cities, Countries
WHERE Cities.country_ISO_code = Countries.country_ISO_code
GROUP BY region
-- group by an orderable Java data type
SELECT city, AVG(rainfall)
FROM ManyCities
GROUP BY city
HAVING clause

A HAVING clause restricts the results of a GROUP BY in a SelectExpression. The HAVING clause is applied to each group of the grouped table, much as a WHERE clause is applied to a select list. If there is no GROUP BY clause, the HAVING clause is applied to the entire result as a single group.

Syntax

HAVING searchCondition

The searchCondition, which is a specialized booleanExpression, can contain only grouping columns (see “GROUP BY clause” on page 1-80), columns that are part of aggregate expressions, and columns that are part of a subquery. For example, the following query is illegal, because the column C2 is not a grouping column, it does not appear within an aggregate, and it is not within a subquery:

```sql
-- SELECT COUNT(*)
-- FROM t
-- GROUP BY c1
-- HAVING c2 > 1
```

Aggregates in the HAVING clause do not need to appear in the SELECT list. If the HAVING clause contains a subquery, the subquery can refer to the outer query block if and only if it refers to a grouping column.

Example

```sql
-- Find the total number of economy seats taken on a flight,
-- grouped by airline,
-- only when the group has at least 2 records.
SELECT SUM(economy_seats_taken), airline_full
FROM FlightAvailability, Airlines
WHERE FlightAvailability.flight_id.substring(0,2).equals(airline)
GROUP BY airline_full
HAVING COUNT(*) > 1

-- find the average temperature of cities in a region for February
-- where that average is above 75
SELECT AVG (city.showTheTemperature(DATE'1998-02-01', 'F')), region
FROM Cities JOIN Countries USING (country_ISO_code)
GROUP BY region
HAVING AVG (city.showTheTemperature(DATE'1998-02-01', 'F')) > 75.0
```
INNER JOIN

An INNER JOIN is a JOIN operation that allows you to specify an explicit join clause.

Syntax

```
TableExpression [ INNER ] JOIN TableExpression
{
    ON booleanExpression |
    USING (SimpleColumnName [ , SimpleColumnName]* )
}
```

You can specify the join clause in one of two ways:

- Specifying ON with a boolean expression.
- Specifying USING (column list), where every column named in the column list appears in both tables. Cloudscape performs an equijoin between the two tables using the columns named in the column list.

For additional differences between ON and USING clauses, see “ResultSet and Performance Differences Between ON and USING” on page 1-85.

The INNER JOIN operation produces the same results as a query with an equivalent WHERE clause. The following two examples are equivalent:

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

SELECT *
FROM Countries INNER JOIN Cities
USING (country_ISO_code)
```

The scope of expressions in either the ON or the USING clause includes the current tables and any tables in outer query blocks to the current SELECT. In the following example, the ON clause refers to the current tables:

```
SELECT *
FROM Hotels INNER JOIN Cities
ON (Hotels.city_id = Cities.city_id)
```

In the following example, the ON clause appears in a subquery and refers to columns in the outer query:

```
SELECT *
FROM t1
WHERE x IN
    (SELECT y FROM t2 INNER JOIN t3 ON z = t1.a)
```
The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

**Example**

-- join the Flights and FlightAvailability tables
```
SELECT *
FROM Flights JOIN FlightAvailability
USING (flight_id, segment_number)
WHERE orig_airport = 'SFO'
```

-- Join Cities with Countries, use a correlation name for
-- a method invocation
```
SELECT DISTINCT city.getLanguage() AS Lang, country
FROM Cities JOIN Countries
USING (country_ISO_code)
ORDER BY Lang
```

-- Join two values clauses
```
SELECT *
FROM (VALUES (3, 4), (1, 5), (2, 6))
AS valuesTable1(c1, c2)
JOIN (VALUES (3, 2), (1, 2),
      (0, 3)) AS valuesTable2(c1, c2)
ON valuesTable1.c1 = valuesTable2.c1
```

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

-- Join Airlines with a view
```
SELECT airline_full, flight_id, number_seats
FROM Airlines
JOIN Segments_SeatBookings ON
    Segments_SeatBookings.flight_id.substring(0,2).equals(airline)
```

-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
```
SELECT country, city.getName(), hotel_name, normal_rate
FROM COUNTRIES INNER JOIN Cities
INNER JOIN Hotels
ON Countries.country_ISO_code=Cities.country_ISO_code
INNER JOIN Hotels
ON Cities.city_id=Hotels.city_id
```
INNER JOIN

ResultSet and Performance Differences Between ON and USING

In accordance with the ANSI standard, join operations that specify USING return results different from those that specify ON. Consider the following examples:

```
SELECT *
FROM Countries INNER JOIN Cities
ON Countries.country_ISO_code = Cities.country_ISO_code

SELECT *
FROM Countries INNER JOIN Cities
USING (country_ISO_code)
```

The first query returns all the columns in the first table (country, country_ISO_code, and region), plus all the columns in the second table (city_id, city, and country_ISO_code). Note that the join column, country_ISO_code, appears twice in the ResultSet.

For the second query, Cloudscape strips out the join column before returning the result and puts the join column first in the result. The query returns all the columns in the first table (with the join column first), plus all the columns in the second table minus the join column. Thus, the query returns country, country_ISO_code, region, city_id, and city.

Using a select list other than SELECT * allows you to customize your ResultSet.

**NOTE:** Joins with the USING clause have a slightly slower performance than those using ON or those using simple WHERE clauses, because Cloudscape must perform an additional restriction operation to return the final result. For performance reasons, avoid joins with USING clauses.
**INSERT statement**

An INSERT statement creates a row or rows and stores them in the named table.

**Syntax**

```
INSERT INTO TableName
    [ (SimpleColumnName [ , SimpleColumnName]* ) ]
    [ PROPERTIES clause ]
Query
```

*Query* can be:

- a `SelectExpression`
- a `VALUES` list
- a multiple-row `VALUES` expression

Single-row and multiple-row lists can include the keyword DEFAULT. Specifying DEFAULT for a column inserts the column’s default value into the column. Another way to insert the default value into the column is to omit the column from the column list and only insert values into other columns in the table. For more information see “VALUES expression” on page 1-123.

- UNION expressions

For more information about *Query*, see “Query” on page 1-98.

The only valid properties for a PROPERTIES clause in an INSERT statement are `insertMode` property and optionally the `bulkFetch` property. See “Bulk Insert Properties” on page 1-96.

**Example**

```
INSERT INTO Countries
VALUES ('Algeria', 'DZ', 'North Africa')

INSERT INTO SouthAmericanCountries
SELECT *
FROM Countries WHERE region = 'South America'

-- You can insert the results of a subquery into a table,
-- even if the subquery references
-- that table.
-- if DEFAULT values are defined for any non-specified
-- columns, the default values are automatically
-- inserted
INSERT INTO Flights (flight_id, segment_number)
```
The INSERT statement depends on the table being inserted into, all of the conglomerates (units of storage such as heaps or indexes) for that table, and any other table named in the query. Any statement that creates or drops an index or a constraint for the target table of a prepared INSERT statement invalidates the prepared INSERT statement.

The INSERT depends on all aliases used in the query. Dropping aliases invalidates a prepared INSERT statement if the statement uses the alias.

**Interaction with Java Data Types**

Each built-in type in SQL-J has a Java class associated with it. You can insert a value of a built-in type into a column of its corresponding Java class, and vice versa. Table 1-9, “Java Classes Associated with SQL-J Built-In Types,” on page 1-243, shows the correspondence between types.
For example, you can insert an SQL-J INTEGER value into a 
SERIALIZE(java.lang.Integer) column, and you can insert a java.lang.Integer
value into an SQL-J INTEGER column.

When inserting into a column that stores a Java data type, Cloudscape checks
whether the class of the object being inserted is assignable to the type of the column.
If the object’s class is not assignable to the column’s Java data type, Cloudscape
throws the statement exception:

   An attempt was made to put a data value of type "{0}" into a
data value of type "{1}".

For more information, see “Assignability” on page 1-212.
JOIN operation

The JOIN operations, which are among the possible \textit{TableExpressions} in a FROM clause, perform joins between two tables. (You can also perform a join between two tables using an explicit equality test in a WHERE clause, such as \texttt{“WHERE t1.col1 = t2.col2”}.)

Syntax

JOIN operation

The JOIN operations are:

- **CROSS JOIN**
  Specifies a Cartesian product between two tables (no join clause). See “CROSS JOIN” on page 1-62.

- **INNER JOIN**
  Specifies a join between two tables with an explicit join clause. See “INNER JOIN” on page 1-83.

- **LEFT OUTER JOIN**
  Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the first table. See “LEFT OUTER JOIN” on page 1-90.

- **RIGHT OUTER JOIN**
  Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the second table. See “RIGHT OUTER JOIN” on page 1-100.

In all cases, you can specify additional restrictions on one or both of the tables being joined in outer join clauses or in the \textit{WHERE clause}.

JOIN Expressions and Query Optimization

For information on which types of joins are optimized, see \textit{Tuning Cloudscape}.
LEFT OUTER JOIN

A LEFT OUTER JOIN is one of the JOIN operations that allow you to specify a join clause. It preserves the unmatched rows from the first (left) table, joining them with a NULL row in the shape of the second (right) table.

Syntax

```
TableExpression  LEFT OUTER JOIN TableExpression
{                           
   ON booleanExpression  |   
   USING (SimpleColumnName [ , SimpleColumnName]*) 
}
```

The scope of expressions in either the ON or the USING clause includes the current tables and any tables in query blocks outer to the current SELECT. The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does). For additional differences between ON and USING clauses, see “ResultSet and Performance Differences Between ON and USING” on page 1-85.

Examples

```
--match cities to countries, showing even those
--countries that have no matching cities
SELECT country, city.getName()
FROM Countries LEFT OUTER JOIN Cities
USING (country_ISO_code)

-- show all Flights, including those without any seats_booked
SELECT Flights.flight_id, number_seats
FROM Flights
LEFT OUTER JOIN Segments_SeatBookings
ON Flights.flight_id = Segments_SeatBookings.flight_id

-- show multiple join
SELECT country, city.getName(), hotel_name, normal_rate
FROM COUNTRIES LEFT OUTER JOIN Cities
LEFT OUTER JOIN Hotels
ON Countries.country_ISO_code=Cities.country_ISO_code
LEFT OUTER JOIN Hotels
ON Cities.city_id=Hotels.city_id
```
LOCK TABLE statement

Allows a user to explicitly acquire a shared or exclusive table lock on the specified table. The table lock lasts until the end of the current transaction.

Explicitly locking a table is useful for:

- avoiding the overhead of multiple row locks on a table (in other words, user-initiated lock escalation)
- avoiding deadlocks

You cannot lock system tables with this statement.

Syntax

LOCK TABLE TableName IN { SHARE | EXCLUSIVE } MODE

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, a particular statement may 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.

If the specified lock cannot be acquired because another connection already holds a lock on the table, a statement-level exception is raised (*SQLState X0X02*) after the deadlock timeout period.

Examples

```sql
-- lock the entire table in share mode to avoid
-- a large number of row locks
LOCK TABLE Flights IN SHARE MODE;
SELECT *
FROM Flights
WHERE orig_airport > 'OOO';

-- lock the entire table in exclusive mode
-- for a transaction that will update many rows,
-- but where no single statement will update enough rows
-- acquire an exclusive table lock on the table.
-- In a row-level locking system, that transaction would
-- require a large number of locks or might deadlock.
LOCK TABLE HotelAvailability IN EXCLUSIVE MODE;
UPDATE HotelAvailability
SET rooms_taken = (rooms_taken + 2)
WHERE hotel_id = 194 AND booking_date = DATE'1998-04-10';
```
UPDATE HotelAvailability
SET rooms_taken = (rooms_taken + 2)
WHERE hotel_id = 194 AND booking_date = DATE'1998-04-11';

UPDATE HotelAvailability
SET rooms_taken = (rooms_taken + 2)
WHERE hotel_id = 194 AND booking_date = DATE'1998-04-12';

UPDATE HotelAvailability
SET rooms_taken = (rooms_taken + 2)
WHERE hotel_id = 194 AND booking_date = DATE'1998-04-12';

-- if a transaction needs to look at a table before
-- updating it, acquire an exclusive lock before
-- selecting to avoid deadlocks
LOCK TABLE People IN EXCLUSIVE MODE;
SELECT MAX(person_id) + 1 FROM PEOPLE;
-- INSERT INTO PEOPLE . . .
ORDER BY clause

The ORDER BY clause is an optional element of a SELECT statement. An ORDER BY clause allows you to specify the order in which rows appear in the ResultSet.

Syntax

ORDER BY { ColumnName | ColumnPosition }
[ ASC | DESC ]
[ , ColumnName | ColumnPosition
[ ASC | DESC ] ] *

ColumnPosition is an integer that refers to the number of the column in the SelectItem in the underlying Query of the SELECT statement (numbering starts at 1). (In other words, if you want to order by a column, that column must be in the select list.)

ColumnName refers to the names visible from the SelectItems in the underlying query of the SELECT statement. (In other words, if you want to order by a column, that column must be in the select list.)

ASC specifies that the results should be returned in ascending order; DESC specifies that the results should be returned in descending order. If the order is not specified, ASC is the default.

An ORDER BY clause prevents a SELECT statement from being an updatable cursor. (For more information, see “Requirements for Updatable Cursors” on page 1-107.)

For ORDER BY, NULLs sort low. For example, if an INTEGER column contains integers, NULL is considered less than 1 for purposes of sorting.

You cannot ORDER BY a Java data type unless the data type is orderable (see “Orderable Java Data Types” on page 1-214).

NEW: The ability to order by some Java data types is new in Version 3.0.

Example

```
SELECT hotel_name, normal_rate
FROM Hotels
ORDER BY 2 DESC

SELECT city.getName(), city.getDisplayCountry() AS COUNTRY
FROM Cities
ORDER BY COUNTRY
```
-- order by an Orderable Java data type
SELECT *
FROM Cities
ORDER BY City
A PROPERTIES clause, or list of properties, is an optional part of an SQL-J CREATE TABLE statement, CREATE INDEX statement, CONSTRAINT clause, FROM clause, TableExpression, or INSERT statement.

Properties specified in a PROPERTIES clause fall into one of the following categories:

- “Storage-Related Properties” on page 1-95
- “Optimizer Override Properties” on page 1-96
- “Bulk Insert Properties” on page 1-96

**Syntax**

The basic syntax of a PROPERTIES clause is as follows:

```
PROPERTIES propertyName = value [, propertyName = value]*
```

**NOTE:** If you specify the same property multiple times within a PROPERTIES clause, an exception is thrown.

**Storage-Related Properties**

For CREATE TABLE, CREATE INDEX, and CONSTRAINT, a PROPERTIES clause allows you to specify storage properties for the table or index being created and overrides the properties currently set on a database-wide or system-wide basis. These properties include:

- `cloudscape.storage.initialPages`
- `cloudscape.storage.minimumRecordSize`
- `cloudscape.storage.pageSize`
- `cloudscape.storage.pageReservedSpace`

For indexes, you can set only `cloudscape.storage.initialPages` and `cloudscape.storage.pageSize`. For tables, you can set `cloudscape.storage.initialPages`, `cloudscape.storage.minimumRecordSize`, `cloudscape.storage.pageSize`, or `cloudscape.storage.pageReservedSpace`. Any mistyped or unused properties are ignored.

For more information, see “Conglomerate-Specific Properties” on page 1-8 in Tuning Cloudscape.

```
-- set the page size to 262144 and the reserved space to 0%
-- for this table only
```
CREATE TABLE Maps
  (country_ISO_code CHAR(2),
   mapIsBig BOOLEAN,
   map SERIALIZE(JBMSTours.serializabletypes.Picture))
PROPERTIES cloudscape.storage.pageSize=262144,
   cloudscape.storage.pageReservedSpace=0
-- set the page size for the index to 2048
CREATE UNIQUE BTREE INDEX Maps_Unique ON
  maps(country_ISO_code, mapIsBig)
PROPERTIES cloudscape.storage.pageSize = 2048

To verify properties used for a specific table or index, use the static methods of
COM.cloudscape.database.PropertyInfo (aliased as PropertyInfo). For more
information, see “Verifying Conglomerate-Specific Properties” on page 1-9 in the
Cloudscape Developer’s Guide.

Optimizer Override Properties

For a PROPERTIES clause in a FROM clause, you can specify:

• joinOrder

For a PROPERTIES clause in a TableExpression, you can specify for a specific
table:

• bulkFetch
• constraint
• index
• joinStrategy (for the inner table of a join only)

For more information, see Chapter 6, “Optimizer Overrides”, in Tuning
Cloudscape.

Bulk Insert Properties

You can use a PROPERTIES clause to set the insertMode to bulkInsert or replace
in an INSERT statement.

• insertMode=bulkInsert

Setting this property to bulkInsert enables fast load of bulk data if the table
is empty and is not in a target database. The entire table is locked.
NEW: The ability to use fast load of bulk data on a published table in a source database is new in Version 3.0.

If the table being inserted into also appears in the FROM clause, bulkInsert is ignored and Cloudscape performs an ordinary insert.

- insertMode=replace

Setting this property to replace enables fast load of bulk data if the table is not empty, is not a target database, and is not a published table in a source database. The new data replaces any existing data. If you did not set this property, you would have to first delete all the rows from the table, and Cloudscape would log all deletes and would have to delete each from every index on the table. Instead, when you set insertMode to replace, Cloudscape uses minimal logging, and a separate delete step is not required.

The entire table is locked.

If the table in which the data is being replaced also appears in the FROM clause, an exception is raised.

If the table being inserted into has any primary or unique constraint that are referenced by foreign keys in other tables, and those constraints have not been turned off with the SET CONSTRAINTS statement, at the end of the bulk insert (replace), Cloudscape checks the constraints. If any of the foreign key constraints are not satisfied, an exception is thrown, and the statement is rolled back. Any constraints which have been turned off by the user are not checked until the user explicitly turns them back on.

NEW: insertMode=replace is new in Version 3.0.

For more information on importing external data, see the Cloudscape Tools and Utilities Guide.
Query

A query creates a virtual table based on existing tables or constants built into tables.

**Syntax**

```
{ ( Query ) |
  Query UNION [ALL] Query |
  SelectExpression |
  VALUES expression }
```

You can arbitrarily put parentheses around queries, or use the parentheses to control the order of evaluation of UNION operations. UNION operations are evaluated from left to right when no parentheses are present.

You can combine two queries into one using the UNION [ALL] operation. UNION builds an intermediate ResultSet with all of the rows from both queries and eliminates the duplicate rows before returning the remaining rows. UNION ALL returns all rows from both queries as the result.

UNION (not UNION ALL) is not permitted with Java data types unless they are *orderable*. For information, see “Orderable Java Data Types” on page 1-214.

**NEW:** The ability to perform UNION with orderable Java data types is new in Version 3.0.

**Example**

```
-- a Select expression
SELECT *
FROM Hotels

-- a subquery
SELECT *
FROM (SELECT hotel_id FROM Hotels) AS SQ

-- a UNION
-- returns all rows from columns number_kids and number_adults
-- in table Groups
-- and (1,2) and (3,4)
-- number_kids and must be integer columns
SELECT number_kids, number_adults
FROM Groups
UNION ALL
VALUES (1,2), (3,4)
```
VALUES (1, 2, 3)
RIGHT OUTER JOIN

A RIGHT OUTER JOIN is one of the JOIN operations that allow you to specify a JOIN clause. It preserves the unmatched rows from the second (right) table, joining them with a NULL in the shape of the first (left) table. A LEFT OUTER JOIN B is equivalent to B RIGHT OUTER JOIN A, with the columns in a different order.

Syntax

```
TableExpression  RIGHT OUTER JOIN TableExpression
{
    ON booleanExpression  |
    USING (SimpleColumnName  [ , SimpleColumnName]* )
}
```

The scope of expressions in either the ON or the USING clause includes the current tables and any tables in query blocks outer to the current SELECT. The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does). For additional differences between ON and USING clauses, see “ResultSet and Performance Differences Between ON and USING” on page 1-85.

Examples

```
-- get all countries and corresponding cities, including
-- countries without any cities
SELECT city.getName(), country
FROM Cities RIGHT OUTER JOIN Countries
ON Cities.country_ISO_code = Countries.country_ISO_code

-- get all countries in Africa and corresponding cities, including
-- countries without any cities
SELECT city, country
FROM Cities RIGHT OUTER JOIN Countries
USING (country_ISO_code)
WHERE Countries.region = 'Africa'

-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
SELECT country, city.getName(), hotel_name, normal_rate
FROM Cities RIGHT OUTER JOIN Countries
ON Countries.country_ISO_code=Cities.country_ISO_code
INNER JOIN Hotels
ON Cities.city_id=Hotels.city_id
```
ScalarSubquery

You can place a ScalarSubquery anywhere an Expression is permitted. A ScalarSubquery turns a SelectExpression result into a scalar value because it returns only a single row and column value.

The query must evaluate to a single row with a single column.

Sometimes also called an expression subquery.

Syntax

(Query)

Example

-- avg always returns a single value, so the subquery is
-- a ScalarSubquery
SELECT group_id FROM Groups
WHERE budget-running_total >
  (SELECT avg(budget-running_total) FROM Groups)

-- another ScalarSubquery
SELECT hotel_name
FROM Hotels
WHERE city_id = 40 AND normal_rate <
  (SELECT budget-running_total
   FROM Groups WHERE group_id = 1)
SelectExpression

A SelectExpression is the basic SELECT-FROM-WHERE construct used to build a table value based on filtering and projecting values from other tables.

Syntax

```
SELECT [ DISTINCT | ALL ] SelectItem [ , SelectItem ]*
FROM clause
[ WHERE clause ]
[ GROUP BY clause ]
[ HAVING clause ]
```

SelectItem

```
{
  * |
  { TableName | CorrelationName } .* | 
  Expression [AS SimpleColumnName ]
}
```

The SELECT clause contains a list of expressions and an optional quantifier that is applied to the results of the FROM clause and the WHERE clause. If DISTINCT is specified, only one copy of any row value is included in the result. Nulls are considered duplicates of one another for the purposes of DISTINCT. If no quantifier, or ALL, is specified, no rows are removed from the result in applying the SELECT clause (ALL is the default).

A SelectItem projects one or more result column values for a table result being constructed in a SelectExpression.

The result of the FROM clause is the cross product of the FROM items. The WHERE clause can further qualify this result.

The WHERE clause causes rows to be filtered from the result based on a boolean expression. Only rows for which the expression evaluates to TRUE are returned in the result.

The GROUP BY clause groups rows in the result into subsets that have matching values for one or more columns. GROUP BY clauses are typically used with aggregates.

If there is a GROUP BY clause, the SELECT clause must contain only aggregates or grouping columns. If you want to include a non-grouped column in the SELECT clause, include the column in an aggregate expression. For example:

```
-- to find the number of hotels in each city
-- if you want to see the city name, you can include it in an
-- aggregate expression
```
SELECT COUNT(*), MAX(city.getName())
FROM Cities, Hotels
WHERE Cities.city_id = Hotels.city_id
GROUP BY Hotels.city_id

If there is no GROUP BY clause, but a SelectItem contains an aggregate not in a
subquery, the query is implicitly grouped. The entire table is the single group.

The HAVING clause restricts a grouped table, specifying a search condition (much
like a WHERE clause) that can refer only to grouping columns or aggregates from
the current scope. The HAVING clause is applied to each group of the grouped
table. If the HAVING clause evaluates to TRUE, the row is retained for further
processing. If the HAVING clause evaluates to FALSE or NULL, the row is
discarded. If there is a HAVING clause but no GROUP BY, the table is implicitly
grouped into one group for the entire table.

Cloudscape processes a SelectExpression in the following order:

- FROM clause
- WHERE clause
- GROUP BY (or implicit GROUP BY)
- HAVING clause
- SELECT clause

The result of a SelectExpression is always a table.

When a query does not have a FROM clause (when you are constructing a value,
not getting data out of a table), you use a VALUES statement, not a
SelectExpression. For example:

VALUES CURRENT_TIMESTAMP

See “VALUES expression” on page 1-123.

**The * Wildcard**

* expands to all columns in the tables in the associated FROM clause.

TableName.* and CorrelationName.* expand to all columns in the identified table.
That table must be listed in the associated FROM clause.

**Naming Columns**

You can name a SelectItem column using the AS clause. When the SelectExpression
appears in a UNION, the names from the first SelectExpression are taken as the
names for the columns in the result of the UNION. If a column of a SelectItem is not a simple ColumnReference expression or named with an AS clause, it is given a generated unique name.

These column names are useful in several cases:

- They are made available on the JDBC ResultSetMetaData.
- They are used as the names of the columns in the resulting table when the SelectExpression is used as a table subquery in a FROM clause.
- They are used in the ORDER BY clause as the column names available for sorting.

Java Data Types

DISTINCT and GROUP BY require sorting and so are not permitted with Java data types unless they are orderable. For information, see “Orderable Java Data Types” on page 1-214.

NEW: The ability to perform DISTINCT and GROUP BY with orderable Java data types is new in Version 3.0.

Examples

-- this example shows SELECT-FROM-WHERE
-- with an ORDER BY clause
-- and correlationNames for the tables
SELECT CONSTRAINTNAME, COLUMNNAME
FROM SYS.SYSTABLES t, SYS.SYSCOLUMNS col,
SYS.SYSCONSTRAINTS cons, SYS.SYSCHECKS checks
WHERE t.TABLENAME = 'FLIGHTS'
AND t.TABLEID = col.REFERENCEID
AND t.TABLEID = cons.TABLEID
AND cons.CONSTRAINTID = checks.CONSTRAINTID
AND REFERENCEDCOLUMNS.isReferencedColumn(col.COLUMNNUMBER)
ORDER BY CONSTRAINTNAME

-- This example shows the use of the DISTINCT clause
SELECT DISTINCT normal_rate
FROM Hotels

-- This example shows how to rename an expression
SELECT MAX(city.showTemperature()) AS HottestTemp, Region
FROM Cities, Countries
WHERE Cities.country_ISO_code = Countries.country_ISO_ISO_CODE
GROUP BY Region
ORDER BY HottestTemp
SELECT statement

A SELECT statement consists of a query with an optional ORDER BY clause, an optional FOR UPDATE clause, and an optional AT ISOLATION clause. The SELECT statement is so named because the typical first word of the query construct is SELECT. (Query includes the VALUES expression and UNION expressions as well as SELECT expressions).

The ORDER BY clause guarantees the ordering of the ResultSet. The FOR UPDATE clause makes the result an updatable cursor. The AT ISOLATION clause allows the user to specify an isolation level for a specific query without committing the current transaction.

Syntax

Query
[ORDER BY clause]
[FOR UPDATE clause]
[AT ISOLATION clause]

Example

-- a query with an ORDER BY clause
SELECT city.getDistanceFrom(
    findCity(getCurrentConnection(), 35))
    AS distance_from_paris, city.getName() AS name
FROM Cities
ORDER BY distance_from_paris

-- creating an updatable cursor with a FOR UPDATE clause
SELECT hotel_id, booking_date, rooms_taken
FROM HotelAvailability WHERE hotel_id =3 AND
EXTRACT(MONTH FROM booking_date) = 9
FOR UPDATE OF rooms_taken

-- set the isolation level to SERIALIZABLE
SELECT *
FROM FlightAvailability
WHERE flight_id BETWEEN 'AA1000' AND 'AA1200'
AT ISOLATION SERIALIZABLE

A SELECT statement returns a ResultSet. A cursor is a pointer to a specific row in ResultSet. In Java applications, all ResultSets are cursors. A cursor is updatable—that is, you can update or delete rows as you step through the ResultSet—if the SELECT statement that generated it and its underlying query meet cursor...
updatability requirements, as detailed below. You use a FOR UPDATE clause when
you want to generate an updatable cursor.

**NOTE:** The ORDER BY clause allows you to order the results of the SELECT.
Without the ORDER BY clause, the results are returned in random order.
No order is guaranteed without the ORDER BY clause. ORDER BY is not
permitted with non-orderable Java data types.

If a SELECT statement meets the requirements listed below, cursors are updatable
only if you specify FOR UPDATE in the FOR clause (see “FOR UPDATE clause”
on page 1-78).

### Requirements for Updatable Cursors

Only simple, single-table SELECT cursors can be made updatable. To generate
updatable cursors:

- The SELECT statement must not include an ORDER BY clause.
- The underlying *Query* must be a *SelectExpression*.
- The *SelectExpression* in the underlying *Query* must not include:
  - DISTINCT
  - Aggregates
  - GROUP BY clause
  - HAVING clause
- The FROM clause in the underlying *Query* must not have:
  - more than one table in its FROM clause
  - anything other than one table name
  - *SelectExpressions*
  - subqueries

There is no SQL-J language statement to assign a name to a cursor. Instead, you
use the JDBC API to assign names to cursors or retrieve system-generated names.
For more information, see “Naming or Accessing the Name of a Cursor” on page
6-9 in the *Cloudscape Developer’s Guide*.

Cursors are read-only by default. For a cursor to be updatable, you must specify
FOR UPDATE in the FOR clause (see “FOR UPDATE clause” on page 1-78).
Dependency System

The SELECT depends on all the tables and views named in the query and the conglomerates (units of storage such as heaps and indexes) chosen for access paths on those tables. CREATE INDEX does not invalidate a prepared SELECT statement. A DROP INDEX statement invalidates a prepared SELECT statement if the index is an access path in the statement. If the SELECT includes views, it also depends on the dictionary objects on which the view itself depends (see “CREATE VIEW statement” on page 1-60).

Any prepared UPDATE WHERE CURRENT or DELETE WHERE CURRENT statement against a cursor of a SELECT depends on the SELECT. Removing a SELECT through a java.sql.Statement.close request invalidates the UPDATE WHERE CURRENT or DELETE WHERE CURRENT.

The SELECT depends on all aliases used in the query. Dropping an alias invalidates a prepared SELECT statement if the statement uses the alias.
SET CONSTRAINTS statement

The SET CONSTRAINTS statement allows you to temporarily disable or re-enable a specific foreign key or check constraint or constraints, all foreign key or check constraints in a table, or all foreign key and check constraints in a database. In addition, you are allowed to specify a primary key or unique constraint. Disabling a foreign key or check constraint actually disables the constraint. “Disabling” a primary or unique constraint disables all the foreign keys that reference that constraint and does not actually turn off the primary or unique constraint.

Syntax

SET CONSTRAINTS
{   ALL | ConstraintName [, ConstraintName ]* | FOR TableName }
} { ENABLED | DISABLED }

Re-enabling a constraint requires that every row in the table meet the constraint condition. If a row violates the constraint, the constraint remains in DISABLED mode, and an exception is thrown. If a single SET CONSTRAINTS command affects more than one constraint and any single constraint fails, no constraints are re-enabled, and an exception is thrown.

Examples

-- turn off the foreign-key and check constraints
-- in the HotelAvailability table
SET CONSTRAINTS
FOR HotelAvailability
DISABLED

-- turn them all back on
SET CONSTRAINTS
ALL
ENABLED

-- turn off all foreign keys that reference the Cities
-- primary key
-- CITIES_PK will be marked as disabled in
-- the SYS.SYSCONSTRAINTS system table
SET CONSTRAINTS
CITIES_PK
DISABLED
The state of a constraint (whether it is enabled or disabled) is reflected in the `STATE` column of `SYS.SYSCONSTRAINTS`. Primary and unique keys that have been “disabled” are marked disabled in this table, even though only foreign keys that reference them are actually disabled.
**SET RUNTIMESTATISTICS statement**

You can turn RunTimeStatistics on or off for a connection with the SET RUNTIMESTATISTICS statement. By default, RunTimeStatistics are off. When RunTimeStatistics is on, Cloudscape creates an object that implements the `COM.cloudscape.types.RunTimeStatistics` interface for each statement executed within the connection (except for commit) until the attribute is turned off.

For statements that do not return rows, the object is created when all internal processing has completed before returning to the client program. For statements that return rows, the object is created when the first `next()` call returns 0 rows or a `close()`, whichever comes first.

You access the object created with the `RUNTIMESTATISTICS()` built-in function. See “`RUNTIMESTATISTICS()`” on page 1-161.

**Syntax**

```
SET RUNTIMESTATISTICS { ON | OFF }
```

**Example**

```
-- establish a connection first
-- turns on RUNTIMESTATISTICS
SET RUNTIMESTATISTICS ON
-- for connection
-- execute complex query here
-- step through the result sets
-- access runtime statistics information here
SET RUNTIMESTATISTICS OFF
```
SET SCHEMA statement

The SET SCHEMA statement sets the default schema for a connection’s session to the designated schema. The default schema is used as the target schema for all statements issued from the connection that do not explicitly specify a schema name.

The target schema must exist for the SET SCHEMA statement to succeed. See “CREATE SCHEMA statement” on page 1-44.

If the schema is not specified, the default schema is set to the APP schema unless a user name was specified upon connection. If a user name was specified, the user’s name is the default schema for the connection if a schema with that name exists. If one does not exist, the user defaults to the APP schema.

The SET SCHEMA statement is not transactional: If the SET SCHEMA statement is part of a transaction that is rolled back, the schema change remains in effect.

Syntax

SET SCHEMA SchemaName

Examples

-- Create some objects in the hotel schema
SET SCHEMA hotel

-- Create a table in the hotel schema
CREATE TABLE names (name VARCHAR(20))

-- Move back to the APP schema
SET SCHEMA APP

-- Get data from the hotel schema
SELECT * FROM hotel.names
SET STATISTICS TIMING statement

Statistics timing is an attribute associated with a connection that you turn on and off via the SET STATISTICS statement. Statistics timing is off by default. Turn statistics timing on only when RunTimeStatistics is already on (see “SET RUNTIMESTATISTICS statement” on page 1-111). Turning it on when RunTimeStatistics is off has no effect.

When statistics timing is turned on, Cloudscape tracks the timings of various aspects of the execution of a statement. This information is included in the COM.cloudscape.types.RunTimeStatistics object returned by RUNTIMESTATISTICS(). (See “RUNTIMESTATISTICS()” on page 1-161.) Statistics timing is a separate attribute because the cost of obtaining system time may be relatively expensive. When statistics timing is off, RUNTIMESTATISTICS() shows all timings as zeroes.

Syntax

SET STATISTICS TIMING { ON | OFF }

Example

SET RUNTIMESTATISTICS ON
SET STATISTICS TIMING ON
SET TRANSACTION ISOLATION LEVEL statement

The SET TRANSACTION ISOLATION LEVEL statement allows a user to change the isolation level for the user’s connection. Valid levels are SERIALIZABLE and READ COMMITTED.

Issuing this command commits the current transaction, which is consistent with the java.sql.Connection.setTransactionLevel method. To change the isolation without committing the current transaction, see “AT ISOLATION clause” on page 1-24.

For information about isolation levels, see “Locking, Concurrency, and Isolation” on page 6-13 in the Cloudscape Developer’s Guide.

Syntax

SET TRANSACTION ISOLATION LEVEL
{
  READ COMMITTED | SERIALIZABLE
}

Example

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
SET TRIGGERS statement

SET TRIGGERS allows you to disable and to re-enable a specified trigger or triggers, all triggers for a specified table, or all triggers in the database. (Re-enabling a trigger does not cause it to fire on any changes that were made while it was disabled.)

Syntax

SET TRIGGERS
{
    ALL |
    TriggerName [, TriggerName ] * } |
    FOR TableName
}
{ ENABLED | DISABLED }

ALL disables or re-enables all triggers in the database that exist at the time the SET TRIGGERS statement is executed should be disabled or re-enabled. FOR TableName enables or re-enables all triggers on the table that exist at the time the statement is executed.

Disabling or re-enabling a trigger is persistent and affects all connections.

Example

SET TRIGGERS FOR Hotels DISABLED
TableExpression

{  
  { TableOrViewExpression | VirtualTableExpression } | ExternalVirtualTable |
  JOIN operation
  
}

TableOrViewExpression

{ TableName | ViewName }  
  [ [ AS ] CorrelationName  
  [ ( SimpleColumnName [ , SimpleColumnName]* ) ] ]  
  [ PROPERTIES clause ]

VirtualTableExpression

TableSubquery  
  [ AS ] CorrelationName  
  [ ( SimpleColumnName [ , SimpleColumnName]* ) ]  
  [ PROPERTIES clause ]

A TableExpression specifies a real table or view or a virtual table in a FROM clause. It is the source from which a SelectExpression selects a result. Virtual tables that are not ExternalVirtualTables (TableSubqueries, including VALUES expressions) are delimited by parentheses (see “VALUES expression” on page 1-123 and “TableSubquery” on page 1-119) and must have correlation names. It also has an optional PROPERTIES clause that allows you to override the optimizer’s choice of access path, join strategy, and I/O size (bulk fetch size).

A correlation name can be applied to a table in a TableExpression so that its columns can be qualified with that name. If you do not supply a correlation name, the table name qualifies the column name. When you give a table a correlation name, you cannot use the table name to qualify columns. You must use the correlation name when qualifying column names.

ExternalVirtualTables require correlation names. See the syntax for “ExternalVirtualTable” on page 1-76 for more information.

No two items in the FROM clause can have the same correlation name, and no correlation name can be the same as an unqualified table name specified in that FROM clause.

In addition, you can give the columns of the table new names in the AS clause. Some situations in which this is useful:

- When a VALUES expression is used as a TableSubquery, since there is no other way to name the columns of a VALUES expression.
When column names would otherwise be the same as those of columns in other tables; renaming them means you don’t have to qualify them.

To make its name match that of a column in another table so that you can use a JOIN operation with the USING clause.

The Query in a TableSubquery appearing in a FromItem can contain multiple columns and return multiple rows. See “TableSubquery” on page 1-119.

The PROPERTIES clause allows you to override the optimizer. Legal properties are:

- bulkFetch
- constraint
- index
- joinStrategy

NEW: Beginning in Version 3.0, you can specify a joinStrategy PROPERTIES clause for joins in which a virtual table expression is the inner table.

For information about the optimizer overrides you can specify, see Tuning Cloudscape.

With the exception of joinStrategy, properties can be specified only on base tables and are not allowed on views or TableExpressions in a FROM clause (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. Since most error checking for the PROPERTIES clause is done during query optimization, errors for PROPERTIES clauses in a CREATE VIEW statement are not returned until the view is first used in a SELECT statement.

Examples

```sql
-- SELECT from a table subquery that has a correlation name
-- (a derived table)
SELECT VirtualFlightTable.flight_number
FROM (SELECT flight_ID, orig_airport, dest_airport
      FROM Flights WHERE orig_airport = 'SFO'
      OR dest_airport = 'SCL' )
AS VirtualFlightTable (flight_number, airport1, airport2)

-- SELECT from a Join expression
SELECT Cities.city_id, Hotels.hotel_id
FROM Cities JOIN Hotels USING (city_id)
WHERE Cities.city_id < 20
```
-- SELECT from an ExternalVirtualTable
-- you must give ExternalVirtualTables correlation names
SELECT *
FROM NEW ExternalQuery(
    'jdbc:cloudscape:History', 'SELECT * FROM HotelBookings')
AS EQ

-- force use of the primary key constraint
-- instead of another index that includes
-- the flight_id column
SELECT *
FROM Flights
PROPERTIES constraint= Flights_pk
WHERE flight_id LIKE 'AA111%'

-- force a hash join
SELECT a.name FROM NEW jarvti('lib/cloudscape.jar') AS a,
    NEW jarvti('lib/tools.jar') AS b
    PROPERTIES joinStrategy=hash
WHERE a.name=b.name
**TableSubquery**

A *TableSubquery* is a subquery that returns multiple rows.

Unlike a *ScalarSubquery*, a *TableSubquery* is allowed only:

- as a *TableExpression* in a FROM clause
- with EXISTS, IN, or quantified comparisons (see “Quantified comparison” on page 1-225)

When used as a *TableExpression* in a FROM clause, it can return multiple columns. When used with EXISTS, it returns multiple columns only if you use * to return the multiple columns.

When used with IN or Quantified comparisons, it must return a single column.

**Syntax**

```sql
(Query)
```

**Example**

-- a subquery used as a TableExpression in a FROM clause
```
SELECT VirtualFlightTable.flight_ID
FROM
   (SELECT flight_ID, orig_airport, dest_airport
      FROM Flights
      WHERE (orig_airport = 'SFO' OR dest_airport = 'SCL') )
AS VirtualFlightTable
```

-- a subquery (values expression) used as a TableExpression
-- in a FROM clause
```
SELECT mycol1
FROM
   (VALUES (1, 2), (3, 4))
AS mytable (mycol1, mycol2)
```

-- a subquery used with EXISTS
```
SELECT *
FROM Flights
WHERE EXISTS
   (SELECT * FROM Flights WHERE dest_airport = 'SFO'
    AND orig_airport = 'GRU')
```

-- a subquery used with IN
```
SELECT flight_id, segment_number
FROM Flights
WHERE flight_id IN
```

*Cloudscape Version 3.0*
(SELECT flight_ID
    FROM Flights WHERE orig_airport = 'SFO'
    OR dest_airport = 'SCL')

-- a subquery used with a quantified comparison
SELECT *
FROM Hotels
WHERE normal_rate < ALL
    (SELECT budget/350 FROM Groups)
**UPDATE statement**

An UPDATE statement sets the value in a column.

You can update the current row of an open, updatable cursor. If there is no current row, or if the current row no longer satisfies the cursor’s query, an exception is raised.

**Syntax**

```
{ 
  UPDATE TableName
  SET ColumnName = Value
    [ , ColumnName = Value] *
  [WHERE clause] | 
  UPDATE [TableName]
  SET ColumnName = Value
    [ , ColumnName = Value ]*
  WHERE CURRENT OF clause
}
```

**Value**

`Expression | DEFAULT`

The first syntactical form is called a searched update. The second syntactical form is called a positioned update.

For searched updates, you update all rows of the table for which the WHERE clause evaluates to TRUE.

For positioned updates, you can update only columns that were included in the FOR UPDATE clause of the SELECT statement that created the cursor. If the SELECT statement did not include a FOR UPDATE clause, the columns must be in the target table of the SELECT statement.

Specifying DEFAULT for the update value sets the value of the column to the default defined for that table.

**Example**

```
UPDATE Airlines
SET basic_rate = basic_rate * 1.2
WHERE Airline = ‘US’

UPDATE HOTELS
SET hotel_name = ?
WHERE hotel_id = ?
```
A searched update statement depends on the table being updated, all of its conglomerates (units of storage such as heaps or indexes), all of its constraints, and any other table named in the WHERE clause or SET expressions. A CREATE or DROP INDEX statement or an ALTER TABLE statement for the target table of a prepared searched update statement invalidates the prepared searched update statement.

The positioned update statement depends on the cursor and any tables the cursor references. You can compile a positioned update even if the cursor hasn’t been opened yet. However, removing the open cursor with the JDBC close method invalidates the positioned update.

A CREATE or DROP INDEX statement or an ALTER TABLE statement for the target table of a prepared positioned update invalidates the prepared positioned update statement.

Both searched and positioned update statements depend on all aliases used in the query. Dropping an alias invalidates a prepared update statement if the latter statement uses the alias.

Dropping or adding triggers on the target table of the update invalidates the update statement.

**Interaction with Java Data Types**

The values in columns that store Java data types are altered only when they are the target columns of an UPDATE statement. Altering the object itself by calling a Java method does not write that change to the database.
VALUES expression

The VALUES expression allows construction of a row or a table from other values. You use a VALUES statement when you do not have a FROM clause. This construct can be used in all the places where a query can, and thus can be used as a statement that returns a ResultSet, within expressions and statements wherever subqueries are permitted, and as the source of values for an INSERT statement.

A common use of the VALUES clause is to call a static method that returns a value or to call a method on an object not serialized in the database that returns a value.

Syntax

```
{  
    VALUES ( Value {, Value }* )  
    |  
    VALUES Value [ , Value ]* |  
    VALUES Work Unit Invocation  
}
```

Value

Expression | DEFAULT
The first form constructs multi-column rows. The second form constructs single-column rows, each expression being the value of the column of the row.

The DEFAULT keyword is allowed only if the VALUES expression is in an INSERT statement. Specifying DEFAULT for a column inserts the column’s default value into the column. Another way to insert the default value into the column is to omit the column from the column list and only insert values into other columns in the table.

In a Cloudscape synchronization system, you can use the third form of the VALUES clause to invoke a work unit, a type of method alias registered with the system in which only the method call and target parameter values are replicated, not any underlying statements. The parameters are applied at the target when VALUES is applied at the target, and they are saved and sent to the source for use during the next refresh. For more information about work unit invocation, see the Cloudscape Synchronization Guide.

Example

```
-- constructing a derived table  
VALUES ('orange', 'orange'), ('apple', 'red'),  
('banana', 'yellow')
```
-- inserting multiple rows of multiple columns into a table
INSERT INTO FruitsAndColors (fruit, color)
VALUES ('orange', 'orange'), ('apple', 'red'),
('banana', 'yellow')

-- insert a row with a DEFAULT value for a column
INSERT INTO HotelAvailability
VALUES (110, current_date, DEFAULT)

-- using a built-in function
VALUES CURRENT_DATE

-- getting the value of an arbitrary expression
VALUES (3*29, 26.0E0/3)

-- getting the value returned by a static method
VALUES (CLASS java.lang.Integer).toString(1)

-- calling a method on an object not serialized in the database
VALUES 'Santiago'.toUpperCase()

-- calling a method on an object not serialized in the database
-- if the method does not return a value, use the CALL statement
VALUES NEW JBMSTours.CreateSchema().createSQLScript(
    getCurrentConnection())

-- calling a method alias that returns a value
VALUES findCity(getCurrentConnection(), 'Paris', 'France')

-- at next refresh, work unit invocation is replicated at source,
-- along with parameter values
-- you can access the value returned by the work unit
VALUES myWorkUnit(getCurrentConnection(), ?, ?)
WHERE clause

A WHERE clause is an optional part of a SelectExpression, DELETE statement, or UPDATE statement. The WHERE clause lets you select rows based on a boolean expression. Only rows for which the expression evaluates to TRUE are returned in the result, or, in the case of a DELETE statement, deleted, or, in the case of an UPDATE statement, updated.

Syntax

WHERE Boolean expression

SQL-J extends the SQL-92 WHERE clause to allow any boolean expression. Most of the general expressions listed in Table 1-7, “Table of Expressions,” on page 1-217, can result in a boolean value. In addition, SQL-J allows the following as expressions in a WHERE clause:

- the TRUE and FALSE constants
- method invocations that return the Java boolean or java.lang.Boolean types
- boolean columns
- subqueries that return boolean expressions

For example, the following are valid SQL-J WHERE clauses:

WHERE TRUE
WHERE tab.userTypeColumn.methodReturningBoolean(17, abcd)
WHERE tab.booleanColumn
WHERE (CLASS COM.ACMEWeb.WebPage).staticBooleanMethod()
WHERE tab.userTypeColumn->booleanField
WHERE (CLASS COM.ACMEWeb.WebPage)->staticBooleanField

In addition, there are the more common boolean expressions. Specific SQL-J boolean operators listed in Table 1-8, “SQL-J Boolean Operators,” on page 1-223, take one or more operands; the expressions return a boolean value.

Examples

WHERE DATE'1993-01-01' < DATE'1996-01-01' -- returns true
WHERE 'This value is here' LIKE '%value_is%' -- evaluates to true

-- find the flights where no business-class seats have been booked
SELECT * FROM FlightAvailability
WHERE business_seats_taken IS NULL
OR business_seats_taken = 0

SELECT *
FROM HotelBookings
WHERE arrival BETWEEN DATE'1999-01-01' AND CURRENT_DATE

-- update only specific rows
UPDATE FlightAvailability
SET business_seats_taken = ((business_seats_taken IS NULL) ?
0: business_seats_taken) +1
WHERE flight_id = 'AA1290'
and segment_number = 1

SELECT Part
FROM Orders O, Suppliers OS
WHERE O.SupplierName = OS.Company
AND EXISTS (SELECT *
FROM Suppliers S
WHERE S.Company <> OS.Company
AND S.Part = OS.Part)
WHERE CURRENT OF clause

The WHERE CURRENT OF clause is a clause in some UPDATE and DELETE statements. It allows you to perform positioned updates and deletes on updatable cursors. For more information about updatable cursors, see “SELECT statement” on page 1-106.

Syntax

WHERE CURRENT OF CursorName

Example

Statement s = conn.createStatement();
s.setCursorName("AirlinesResults");
ResultSet rs = conn.executeQuery(
   "SELECT Airline, basic_rate " +
   "FROM Airlines FOR UPDATE OF basic_rate");
Statement s2 = conn.createStatement();
s2.executeUpdate("UPDATE Airlines SET basic_rate = basic_rate " +
   "+ .25 WHERE CURRENT OF AirlinesResults");
Built-In Functions

A built-in function is an expression in which an SQL-J keyword or special operator executes some operation. Cloudscape lets you execute any number of operations by invoking Java methods in the prescribed manner (see “Method Invocation” on page 1-231). Java method names are case-insensitive. Built-in functions, by contrast, do not use the method invocation syntax, but instead use keywords or special built-in operators. Built-ins are SQL92Identifiers and are thus case-insensitive.

Standard Built-In Functions

- “BIT_LENGTH” on page 1-133
- “CAST” on page 1-134
- “CHAR_LENGTH, CHARACTER_LENGTH” on page 1-143
- “Concatenation” on page 1-144
- “Conditional (?:)” on page 1-145
- “CURRENT_DATE” on page 1-148
- “CURRENT_TIME” on page 1-149
- “CURRENT_TIMESTAMP” on page 1-150
- “CURRENT_USER” on page 1-151
- “EXTRACT” on page 1-152
- “LOWER” on page 1-153
- “LTRIM” on page 1-154
- “OCTET_LENGTH” on page 1-159
- “RTRIM” on page 1-160
- “RUNTIMESTATISTICS()” on page 1-161
- “SESSION_USER” on page 1-169
- “SUBSTRING” on page 1-170
- “SUBSTR” on page 1-172
- “TRIM” on page 1-175
- “UPPER” on page 1-177
- “USER” on page 1-178

Some built-in functions use SQL-J keywords, but, because of their relationship to Java, are described in “Java Expressions” on page 1-230.
You can make any static Java method look and act like a built-in function by creating an alias for it. For more information, see “CREATE METHOD ALIAS statement” on page 1-43.

**Aggregates (Set Functions)**

This section describes aggregates (also described as *set functions* in ANSI SQL-92 and as *column functions* in some database literature). They provide a means of evaluating an expression over a set of rows. Whereas the other built-in functions operate on a single expression, aggregates operate on a set of values and reduce them to a single scalar value. Built-in aggregates can calculate the minimum, maximum, sum, count, and average of an expression over a set of values as well as count rows. You can also create your own aggregates to perform other set functions such as calculating the standard deviation.

**NEW:** User-defined aggregates are new in Version 3.0.

The built-in aggregates can operate on the data types shown in Table 1-2.

**Table 1-2 Permitted Data Types for Built-in Aggregates**

<table>
<thead>
<tr>
<th></th>
<th>All Types</th>
<th>Orderable Java Data Types</th>
<th>Built-in Types (and Corresponding Java Data Types)</th>
<th>Numeric Built-in Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MIN</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Whether user-defined aggregates are allowed on Java data types depends on how you define them.

Aggregates are permitted only in the following:

- A `SelectItem` in a `SelectExpression`.
- A `HAVING` clause.
- An ORDER BY clause (using an alias name) if the aggregate appears in the result of the relevant query block. That is, an alias for an aggregate is permitted in an ORDER BY clause if and only if the aggregate appears in a `SelectItem` in a `SelectExpression`.

If an aggregate appears as a `SelectItem` in a `SelectExpression`, all expressions in `SelectItems` in the `SelectExpression` must be either aggregates or grouped columns.
(see “GROUP BY clause” on page 1-80). (The same is true if there is a HAVING clause without a GROUP BY clause.) This is because the ResultSet of a SelectExpression must be either a scalar (single value) or a vector (multiple values), but not a mixture of both. (Aggregates evaluate to a scalar value, and the reference to a column can evaluate to a vector.) For example, the following query mixes scalar and vector values and thus is not valid:

```sql
-- not valid
SELECT MIN(flying_time), flight_id
FROM Flights
```

Aggregates are not allowed on outer references (correlations). This means that if a subquery contains an aggregate, that aggregate cannot evaluate an expression that includes a reference to a column in the outer query block. For example, the following query is not permitted because SUM operates on a column from the outer query:

```sql
SELECT c1
FROM t1
GROUP BY c1
HAVING c2 >
    (SELECT t2.x
     FROM t2
     WHERE t2.y = SUM(t1.c3))
```

A cursor declared on a ResultSet that includes an aggregate in the outer query block is not updatable.

This section includes the following aggregates:

- “AVG” on page 1-131
- “COUNT” on page 1-146
- “MAX” on page 1-155
- “MIN” on page 1-157
- “SUM” on page 1-174
- “User-Defined Aggregate” on page 1-179
AVG

AVG is an aggregate function that evaluates the average of an expression over a set of rows (see “Aggregates (Set Functions)” on page 1-129). AVG is allowed only on expressions that evaluate to numeric data types.

Syntax

\[
\text{AVG} \left( \text{[ DISTINCT | ALL ] Expression } \right)
\]

The DISTINCT qualifier eliminates duplicates. The ALL qualifier retains duplicates. ALL is the default value if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1.0, 1.0, 1.0, 1.0, and 2.0, AVG(col) returns a smaller value than AVG(DISTINCT col).

Only one DISTINCT aggregate expression per SelectExpression is allowed. For example, the following query is not allowed:

\[
\text{SELECT AVG (DISTINCT flying_time), SUM (DISTINCT miles)} \\
\text{FROM Flights}
\]

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to an SQL-92 numeric data type or to a Java data type that is automatically mapped to an SQL-92 numeric data type. You can therefore call methods that evaluate to SQL-92 data types. (For example, a method that returns a java.lang.Integer or int evaluates to an INTEGER.) If an expression evaluates to NULL, the aggregate skips that value.

The resulting data type is the same as the expression on which it operates (it will never overflow). The following query, for example, returns the INT 1 (which may not be what you would expect):

\[
\text{SELECT AVG(c1)} \\
\text{FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1)}
\]

CAST the expression to another data type if you want more precision:

\[
\text{SELECT AVG(CAST (c1 AS DOUBLE PRECISION))} \\
\text{FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1)}
\]

Examples

-- find the average of column 1
\[
\text{SELECT AVG (col1)} \\
\text{FROM (VALUES (1.0, 2), (2.0, 3), (2.0, 3)} \\
\text{AS virtualTable (col1, col2)}
\]
-- find the average flying time of flights between 1000 and 1500 miles
SELECT AVG (flying_time)
FROM Flights
WHERE miles > 1000 AND miles < 1500

-- use AVG in the HAVING clause to filter the groups
SELECT AVG (flying_time), orig_airport
FROM Flights
GROUP BY orig_airport
HAVING AVG (flying_time) > 5.0

-- find the average of a method invocation
SELECT AVG (person.getAge()) AS AVERAGE_AGE
FROM People
BIT_LENGTH

BIT_LENGTH acts on either a character string expression or a bit string expression and returns the number of bits in the result.

Syntax

BIT_LENGTH ( { CharacterExpression | BitExpression } )

Example

-- returns 32
VALUES BIT_LENGTH('zz')

-- returns 8
VALUES BIT_LENGTH(X'55')

-- returns 3
VALUES BIT_LENGTH(B'111')
CAST

CAST converts a value from one data type to another and provides a data type to a dynamic parameter (?) or a NULL value.

CAST expressions are permitted anywhere expressions are permitted.

Syntax

```
CAST ( [ Expression | NULL | ? ]
   AS { DataType | CLASS JavaClassName | ClassAlias } )
```

The data type to which you are casting an expression is the **target type**. The data type of the expression from which you are casting is the **source type**.

Example

```
SELECT *
FROM CustomizedTours
WHERE customized_tour = CAST
( ? AS CLASS JBMSTours.serializabletypes.Tour)
```

CAST Conversions Among SQL-92 Data Types

Figure 1-1 shows valid conversions between sources and targets for SQL-92 data types.
Figure 1-1 Valid CAST conversions between SQL-92 data types

For information about conversions among Java types, see “CAST Conversions of Java Data Types” on page 1-140.

If a conversion is valid, CASTs are allowed. Size incompatibilities between the source and target types may cause runtime errors. For example, converting from a boolean to a CHAR is valid. However, a boolean requires at least four characters to be represented as a string. Casting a boolean as a CHAR(3) causes a runtime error.

Notes About CAST Conversions

In this discussion, the Cloudscape SQL-92 data types are categorized as follows:

- **numeric**
  - exact numeric (TINYINT, SMALLINT, INTEGER, LONGINT, DECIMAL, NUMERIC)
  - approximate numeric (FLOAT, REAL, DOUBLE PRECISION)
CAST Conversions to Boolean

Strings and numerics can be converted to booleans.

- For numerics, if the value is 0, the CAST evaluates to false; otherwise, the CAST evaluates to true.
- For bit strings, if the string evaluates to X'0', the CAST evaluates to false; otherwise, the CAST evaluates to true.

```sql
-- returns false
VALUES CAST (X'000' AS boolean);

-- returns true
VALUES CAST (X'001' AS boolean);

-- returns true
VALUES CAST (X'002' AS boolean);
```

- For character strings, “false” (case-insensitive) evaluates to false, and “true” (case-insensitive) evaluates to true. All other strings raise an error (this corresponds to the SQL-92 standard but differs from Java behavior).

A date/time value cannot be converted to a boolean.

Booleans can be compared to any numeric types, and you can assign a numeric type to a boolean and vice versa. See “Comparing Booleans with Numeric Types and Assigning Numeric Types to Booleans” on page 1-188.

CAST and Implicit Conversions to Character Strings

Cloudscape supports implicit conversions of character strings to all the built-in data types and vice-versa. In many situations, you do not need to CAST. For example:

```sql
-- 1 converted to '1'
INSERT INTO charColumn VALUES 1
```
-- 1 converted to '1', booleanColumn converted to string type

\[
\text{SELECT 1 || booleanColumn FROM t1}
\]

For information on when a value is implicitly converted to a String, see “Implicit Type Conversion of String Types” on page 1-186.

Any built-in type can be converted to a character string, provided that the target character string has room to represent the source data type.

The resulting data value conforms to the definition of a literal for the source data type. For example, when a TIME value is converted to a character string, the resulting string is of the format HH:MM:SS. The exception is the BIT data type, which when converted is interpreted as a bit representation of a Unicode character string and is converted directly into the target string. For example, \( \text{CAST(X'0061' AS VARCHAR(10))} \) yields “a” instead of the character string “X'0061'”.

### Size Requirements for Explicit and Implicit Conversions to Character Strings

- Booleans require four characters to represent true and five characters to represent false.
- Numeric values require one character for each digit to the left and each digit to the right of the decimal point and an additional character for negative numbers to represent the minus sign. They also require a character for the decimal point. Approximate numerics may require an additional 3 or 4 characters for the exponent part (3 for a positive exponent, 4 for a negative exponent). For example, DOUBLE PRECISION and FLOAT data types (with a precision greater than or equal to 24) may require the extra characters. Truncation of nonzero digits will raise an exception.

\[
\text{-- returns 1}
\]

\[
\text{VALUES CAST (1.00000 AS CHAR(3))};
\]

- DATE values require 10 characters (8 for digits and 2 for dashes).
- TIME values require 8 characters (6 digits and 2 colons).
- TIMESTAMP fields require 23 characters (1 space, 2 dashes, 2 colons, 1 decimal point, and 17 digits). For TIMESTAMP, the milliseconds portion of the timestamp may be silently truncated if necessary.
- The length required by bit strings depends on the string itself (16 bits per character, using the Unicode character encoding); you can use this formula:
BIT_LENGTH(bitExpression)/16

Padding is performed automatically when the target data type is larger than the source data value. Spaces are used to pad character and date/time values, and zeroes are used for all other types.

Implicit conversion to a string is not supported for the receiver or arguments to a Java method.

CAST and Implicit Conversions from Character Strings

Cloudscape supports implicit conversions of character strings to all the built-in data types and vice-versa. In many situations you do not need to CAST.

For example:

```sql
-- '1' implicitly converted to 1 when compared to integer
SELECT *
FROM t1
WHERE intColumn = '1'

-- charColumn implicitly converted to boolean type
-- when compared to boolean
SELECT *
FROM t1 booleanColumn = charColumn

-- '1' implicitly converted to int when used in INSERT
INSERT INTO t1 (intColumn) VALUES '1'

-- '2' converted to int
SELECT 1 + '2'
FROM t1

-- charColumn converted to boolean when used as search condition
SELECT *
FROM t1 WHERE charColumn

-- string '1998-01-01' converted to date
INSERT INTO t1 (dateColumn) VALUES '1998-01-01'
```

For information on when a String is implicitly converted to a non-String data type, see “Implicit Type Conversion of String Types” on page 1-186.

For both explicit and implicit conversions, an exception is thrown if the String cannot be converted to the target data type. The basic rules for explicit and implicit conversion from character strings are as follows:
• Invalid characters
A character string can be converted to any data type, but if the string contains an invalid character or invalid formatting, an exception is raised. An invalid character is one that is inappropriate for the target data type. For example, 'hello' cannot be converted to an integer. Formatting that is not consistent with the definition of a literal data value of the target type is invalid. For example, for a string to convert successfully to a TIME data type, the source string must contain three sets of two base-10 digits separated by semicolons. It is not necessary for the string to have the type identifier, though either is acceptable (both TIME'11:11:11' and '11:11:11' yield the same TIME value during conversion). Another example: For a string to convert successfully to an integer data type, it must not contain a decimal point.

```
ij> VALUES 1 + '1.1';
SQLCol1
-----------------
ERROR 22018: Type INTEGER does not recognize the format of the string '1.1'.
ij> VALUES 1 + CAST ('1.1' AS INT);
SQLCol1
-----------------
ERROR 22018: Type INTEGER does not recognize the format of the string '1.1'.
ij> -- implicit conversion of '1.0' to INT fails
ij> VALUES 2 > '1.0';
SQLCol1
-----
ERROR 22018: Type INT does not recognize the format of the string '1.0'.
```

• Leading and trailing spaces
Conversion from a character string to any other noncharacter data type strips out all leading and trailing spaces before the conversion is performed. If the target is a bit string, each character is converted into the corresponding 16-bit Unicode representation, with possible truncation of trailing spaces if they don’t fit into the resulting bit string, or padding with bits set to zero as needed. Character-string-to-character-string conversions retain leading spaces but may truncate or pad trailing spaces as needed.

For information about some additional limitations on implicit conversion, see “Implicit Type Conversion of String Types” on page 1-186.
CAST Conversions from Numeric Types

A numeric type can be converted to any other numeric type. If the target type cannot represent the non-fractional component without truncation, an exception is raised. If the target numeric cannot represent the fractional component (scale) of the source numeric, then the source is silently truncated to fit into the target. For example, casting 763.1234 as INTEGER yields 763.

CAST Conversions from and to Bit Strings

Bit strings can be converted to other bit strings as well as character strings and booleans. Strings that are converted to bit strings are padded with trailing zeros to fit the size of the target bit string.

CAST Conversions of Date/Time Values

A date/time value can always be converted to and from a TIMESTAMPT. If a DATE is converted to a TIMESTAMPT, the TIME component of the resulting TIMESTAMPT is always 00:00:00. If a TIME data value is converted to a TIMESTAMPT, the DATE component is set to the value of CURRENT_DATE at the time the CAST is executed. If a TIMESTAMPT is converted to a DATE, the TIME component is silently truncated. If a TIMESTAMPT is converted to a TIME, the DATE component is silently truncated.

CAST Conversions of Java Data Types

- CAST conversions of Java data types to other Java data types
  You can cast a Java object to a different Java data type following the same rules as those for an explicit cast in Java. Casting up or widening is always safe. Casting down or narrowing may cause an execution error if a Java ClassCastException is thrown while Cloudscape processes the CAST. A CAST that is always illegal causes an error when the statement is prepared.

- CAST conversions of Java data types to SQL-J data types
  You can CAST Java objects to SQL-J data types according to the mapping of Java data types to SQL-J data types (see Table 1-11, “Conversion of Java Types to SQL-J Types,” on page 1-245). If there is a direct mapping of a Java object to an SQL-J data type, conversions allowed on the SQL-J data type are always allowed on the Java object. For example, you can cast a java.lang.Integer to the same data types to which you can cast an INTEGER.
• Casting expressions to Java data types

You can cast a non-NULL expression or dynamic parameter to a Java data type if and only if the expression evaluates to an SQL-J data type that has a direct mapping to the target Java class (see Table 1-10, “Conversion of SQL-J Types to Java Classes During Method Invocation,” on page 1-244). For example, the following expression is legal:

-- 1 evaluates to INTEGER.
-- INTEGER maps directly to java.lang.INTEGER
VALUES CAST (1 AS CLASS java.lang.Integer)

The following expression is not:

-- 1.0 evaluates to DECIMAL.
-- DECIMAL does not map directly to java.lang.INTEGER
VALUES CAST (1.0 AS CLASS java.lang.Integer)

• NULLs and dynamic parameters

You can cast NULL and dynamic parameters to any valid Java class.

Examples

-- return only the integer portion of the mileage
SELECT CAST (miles AS INT)
FROM Flights

-- you can cast expressions other than simple column references,
-- and you can use the CAST expression in an aggregate
SELECT AVG(CAST (customized_tour.getTotalCost() AS INTEGER))
FROM customizedtours

-- when a column stores subclasses,
-- you can use the CAST expression to cast to a subclass
SELECT (CAST (person AS CLASS
JBMSTours.serializabletypes.Child)).getParent()
FROM People
WHERE person INSTANCEOF JBMSTours.serializabletypes.Child

-- convert timestamps to text
INSERT INTO mytable (text_column)
VALUES (CAST (CURRENT_TIMESTAMP AS VARCHAR(100)))

-- you must cast NULL as a data type to use it
SELECT airline
FROM Airlines
UNION ALL
VALUES (CAST (NULL AS CHAR(2)))

-- cast dynamic parameters to specify the correct signature
x.overloadedMethod(CAST (? AS CLASS java.lang.Integer))

-- cast a Java double as a decimal
SELECT CAST (normal_rate AS DECIMAL(5,2))
FROM Hotels

-- cast an INT
VALUES (CAST (12 AS VARCHAR(5)), CAST ('35' AS INT))

ij> -- use a CAST to control the number of digits that appear
ij> SELECT AVG(city.showTemperature())
FROM cities;

<table>
<thead>
<tr>
<th>SQLCol1</th>
</tr>
</thead>
<tbody>
<tr>
<td>69.54545454545&amp;</td>
</tr>
</tbody>
</table>

ij> SELECT CAST (AVG(city.showTemperature()) AS DECIMAL(5,2))
FROM Cities;

<table>
<thead>
<tr>
<th>SQLCol1</th>
</tr>
</thead>
<tbody>
<tr>
<td>69.54</td>
</tr>
</tbody>
</table>

ij> SELECT CAST (AVG(city.showTemperature()) AS INT)
FROM Cities;

<table>
<thead>
<tr>
<th>SQLCol1</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
</tr>
</tbody>
</table>

Cloudscape Reference Manual
CHAR_LENGTH, CHARACTER_LENGTH

CHAR_LENGTH (or its synonym, CHARACTER_LENGTH) is applied to either a character string expression or a bit string expression and returns the number of characters in the result.

Because all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

Syntax

```sql
CHAR[ACTER]_LENGTH ( { CharacterExpression | BitExpression } )
```

Example

```sql
-- returns 20
VALUES CHAR_LENGTH('supercalifragilistic')

-- returns 1
VALUES CHAR_LENGTH(X'FF')

-- returns 3
VALUES CHAR_LENGTH(130)
```
Concatenation

The concatenation operator, ||, concatenates its right operand to the end of its left operand. It operates on a character or bit expression.

Because all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

Syntax

```
{ CharacterExpression  || CharacterExpression  } |
{ BitExpression  || BitExpression  }
```

For character strings, if both the left and right operands are of type CHAR, the resulting type is CHAR; otherwise, it is VARCHAR. The normal blank padding/trimming rules for CHAR and VARCHAR apply to the result of this operator.

The length of the resulting string is the sum of the lengths of both operands.

For bit strings, if both the left and the right operands are of type BIT, the resulting type is BIT; otherwise, it is BIT VARYING.

Example

```
--returns ‘supercalifragilisticexbealidocious(sp?)’
VALUES ‘supercalifragilistic‘ || ‘exbealidocious‘ || ‘(sp?)’

--returns X’baad
VALUES X’ba’ || B’10101101’

-- returns null
VALUES null || ‘AString’

-- returns ‘130asdf’
VALUES 130 || ‘asdf’
```
Conditional (?:)

You can place a conditional expression anywhere an expression is allowed. It chooses an expression to evaluate based on a boolean test.

Syntax

```
BooleanExpression  ?  ThenExpression  :  ElseExpression
```

*ThenExpression* and *ElseExpression* are both expressions that must be type-compatible. For built-in types, this means that the types must be the same or a built-in broadening conversion must exist between the types. For Java data types, this means that the types must be the same or one of the types must be assignable to the other one due to a super/subclass or interface/implementor relationship.

**NEW:** Beginning in Version 3.0, you no longer need to use the conditional expression for avoiding *NullPointerExceptions* when a nullable column becomes a method receiver. If the value of the instance specified in an instance method invocation is null, the result of the invocation is null (SQL-J NULL). However, you still may need to use the conditional expression for when a nullable column is a primitive method parameter; the conditional expression is a good way of doing that.

Example

```
--if there's no matching city_id (null), display a 0
SELECT country, city_id IS NULL? 0 : city_id
FROM Countries LEFT OUTER JOIN Cities
USING (country_ISO_code)

-- If business_seats_taken has any nulls, you'll get an exception
SELECT NEW java.lang.Integer(business_seats_taken)
FROM flightavailability

-- using the conditional expression, you can "convert" any
-- nulls into 0, a valid parameter
SELECT NEW java.lang.Integer(
    (business_seats_taken IS NULL) ? 0: business_seats_taken)
FROM flightavailability
```
COUNT

COUNT is an aggregate function that counts the number of rows accessed in an expression (see “Aggregates (Set Functions)” on page 1-129). COUNT is allowed on all types of expressions, even those that evaluate to Java data types.

Syntax

    COUNT ( [ DISTINCT | ALL ] Expression )

The DISTINCT qualifier eliminates duplicates. The ALL qualifier retains duplicates. ALL is assumed if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1, 1, 1, 1, and 2, COUNT(col) returns a greater value than COUNT(DISTINCT col).

Only one DISTINCT aggregate expression per SelectExpression is allowed. For example, the following query is not allowed:

    -- query not allowed
    SELECT COUNT (DISTINCT flying_time), SUM (DISTINCT miles)
    FROM Flights

An Expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. If an Expression evaluates to NULL, the aggregate is not processed for that value.

The resulting data type is LONGINT.

Example

    -- Count the number of distinct last names AND the total number
    -- of last names
    SELECT COUNT (DISTINCT person.getLastName()),
    COUNT(person.getLastName())
    FROM People

    -- Count the number of countries in each region,
    -- show only regions that have at least 2
    SELECT COUNT (country), region
    FROM Countries
    GROUP BY region
    HAVING COUNT (country) > 1

    -- counting a Java data type
    SELECT COUNT (city)
    FROM Cities
COUNT(*)

COUNT(*) is an aggregate function that counts the number of rows accessed. No NULLs or duplicates are eliminated. COUNT(*) does not operate on an expression.

Syntax

COUNT(*)

The resulting data type is LONGINT.

Example

-- Count the number of rows in the Flights table
SELECT COUNT(*)
FROM Flights

-- Evaluate an expression only if there are
-- rows in the HotelBookings table
SELECT *
FROM HotelAvailability
WHERE EXISTS
  (SELECT COUNT(*)
   FROM HotelBookings)
CURRENT_DATE

CURRENT_DATE returns the current date; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

Syntax

CURRENT_DATE

Example

-- find hotel bookings for the future:

SELECT *
FROM HotelBookings
WHERE arrival > CURRENT_DATE
CURRENT_TIME

CURRENT_TIME returns the current time; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

**Syntax**

```
CURRENT_TIME
```

**Example**

```
VALUES CURRENT_TIME
```
CURRENT_TIMESTAMP

CURRENT_TIMESTAMP returns the current timestamp; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

Syntax

CURRENT_TIMESTAMP

Example

VALUES CURRENT_TIMESTAMP
CURRENT_USER

CURRENT_USER returns the authorization identifier of the current user (the name of the user passed in when the user connected to the database). If there is no current user, it returns APP.

USER, CURRENT_USER, and SESSION_USER are synonyms.

These functions return a string of up to 128 characters.

Syntax

CURRENT_USER

Example

VALUES CURRENT_USER
EXTRACT

EXTRACT extracts the specified field from a date/time expression as an integer or double-precision value.

Syntax

EXTRACT (Field FROM DateTimeExpression)

The fields available for extraction depend on the date/time type that EXTRACT is applied to, as listed in Table 1-3, “Extractable Fields in Date/Time Data Types”.

<table>
<thead>
<tr>
<th>Field</th>
<th>DATE</th>
<th>TIME</th>
<th>TIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MONTH</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DAY</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>HOUR</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MINUTE</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SECOND</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

The result type of EXTRACT is INTEGER except when extracting SECOND from a TIMESTAMP. In the current release, the result type of extracting SECOND from a TIMESTAMP is DOUBLE PRECISION. The result type of extracting SECOND from a TIME is an INT.

Example

```sql
-- returns 1
VALUES EXTRACT(MONTH FROM DATE‘1995-01-01’)
```
LOWERCAP

LOWERCAP takes a character expression as a parameter and returns a string in which all alpha characters have been converted to lowercase.

Because all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

**Syntax**

LOWERCAP ( CharacterExpression )

A CharacterExpression is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string (except a bit expression), or any Java data type that directly maps to a built-in type that is implicitly converted to a string.

If the parameter type is CHAR or LONG VARCHAR, the return type is CHAR or LONG VARCHAR. Otherwise, the return type is VARCHAR.

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

If the CharacterExpression evaluates to null, this function returns null.

**Example**

```sql
-- returns 'asd1#w'
VALUES LOWERCAP('aSD1#w')

-- returns null if the column stores a null value
SELECT LOWERCAP(hotel_name) FROM Hotels

SELECT LOWERCAP(flight_id.toString()) FROM Flights
```
**LTRIM**

LTRIM trims one or more instances of a leading character string, or any characters within the character string, from a character string expression.

```
LTRIM(CharacterExpression [ , trimCharacters ] )
```

A `CharacterExpression` is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string, or any Java data type that directly maps to a built-in type that is implicitly converted to a string.

LTRIM returns NULL if either `CharacterExpression` or `trimCharacters` evaluates to null.

If `trimCharacter` is not specified, it defaults to ‘’ (blank). It can consist of multiple characters.

**Examples**

```
-- returns 'asdf  
VALUES LTRIM(' asdf  ')

--returns 'df'
VALUES LTRIM('asasdf', 'as')

-- returns 'df'
VALUES LTRIM('asdf', 'sa')

-- returns 'asdf'
VALUES LTRIM('asdf', 'fd')

-- returns '0'
VALUES LTRIM(130, 13)
```
MAX

MAX is an aggregate function that evaluates the maximum of the expression over a set of values (see “Aggregates (Set Functions)” on page 1-129). MAX is allowed only on expressions that evaluate to built-in data types (including CHAR, VARCHAR, DATE, TIME, BIT, etc.) or orderable Java data types.

**NEW:** Beginning in Version 3.0, MAX is also allowed on *orderable* Java data types (see “Orderable Java Data Types” on page 1-214).

**Syntax**

```
MAX ( [ DISTINCT | ALL ] Expression )
```

The DISTINCT qualifier eliminates duplicates. The ALL qualifier retains duplicates. These qualifiers have no effect in a MAX expression. Only one DISTINCT aggregate expression per `SelectExpression` is allowed. For example, the following query is not allowed:

```
SELECT COUNT (DISTINCT flying_time), MAX (DISTINCT miles)
FROM Flights
```

The `Expression` can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in data type or to a Java data type that is automatically mapped to a built-in data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a `java.lang.Integer` or `int` evaluates to an INTEGER.) If an expression evaluates to NULL, the aggregate skips that value.

For CHAR, VARCHAR, and LONG VARCHAR, the number of blank spaces at the end of the value may affect how MAX is evaluated. For example, if the values ‘z’ and ‘z ’ are both stored in a column, you cannot control which one will be returned as the maximum, because a blank space has no value.

The resulting data type is the same as the expression on which it operates (it will never overflow).

**Example**

```sql
-- find the latest date in the FlightBookings table
SELECT MAX (travel_date)
FROM FlightBookings

-- find the longest flight originating from each airport,
-- but only when the airport’s longest flight is over 10 hours
SELECT MAX(flying_time), orig_airport
```
FROM Flights
GROUP BY orig_airport
HAVING MAX(flying_time) > 10

-- use MAX on a method invocation
SELECT MAX (customized_tour.getTotalCost().doubleValue())
FROM CustomizedTours

-- use MAX on an orderable Java data type
SELECT MAX(city) FROM Cities
MIN

MIN is an aggregate expression that evaluates the minimum of an expression over a set of rows (see “Aggregates (Set Functions)” on page 1-129). MIN is allowed only on expressions that evaluate to built-in data types (including CHAR, VARCHAR, DATE, TIME, BIT, etc.) or orderable Java data types.

NEW: Beginning in Version 3.0, MIN is also allowed on orderable Java data types (see “Orderable Java Data Types” on page 1-214).

Syntax

\[
\text{MIN ( [ DISTINCT | ALL ] Expression )}
\]

The DISTINCT and ALL qualifiers eliminate or retain duplicates, but these qualifiers have no effect in a MIN expression. Only one DISTINCT aggregate expression per SelectExpression is allowed. For example, the following query is not allowed:

```
SELECT COUNT (DISTINCT flying_time), MAX (DISTINCT miles)
FROM Flights
```

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in data type or to a Java data type that is automatically mapped to a built-in data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a `java.lang.Integer` or `int` evaluates to an INTEGER.) If an expression evaluates to NULL, the aggregate skips that value.

The type’s comparison rules determine the maximum value. For CHAR, VARCHAR, and LONG VARCHAR, the number of blank spaces at the end of the value may affect the result.

The resulting data type is the same as the expression on which it operates (it will never overflow).

Example

-- find the earliest date
```
SELECT MIN (travel_date)
FROM FlightBookings
```

-- find the shortest flight originating from each airport,
-- but only when the airport’s shortest flight is over 5 hours
```
SELECT MIN(flying_time), orig_airport
FROM Flights
```
GROUP BY orig_airport
HAVING MIN(flying_time) > 5

-- use MIN on a method invocation
SELECT MIN(city.showTemperature())
FROM Cities JOIN Countries USING (country_ISO_code)
WHERE region = 'North America'

-- use MIN on an orderable Java data type
SELECT MIN(DISTINCT city) FROM Cities
OCTET_LENGTH

OCTET_LENGTH returns the number of octets in a character string expression or a bit string expression. For a bit string, the number of octets and the number of characters are always the same, since a bit string is considered to be made up of a series of octets.

Syntax

`OCTET_LENGTH ( CharacterExpression | BitExpression )`

A `CharacterExpression` is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string (except a bit expression), or any Java data type that directly maps to a built-in type that is implicitly converted to a string.

Example

```sql
-- returns 20
VALUES OCTET_LENGTH('supercalifragilistic')

-- returns 1
VALUES OCTET_LENGTH(X'FF')
```
RTRIM

RTRIM trims one or more instances of a trailing character string, or any characters within the trailing character string, from a character string expression.

\[
\text{RTRIM} (\text{CharacterExpression} \ [\ , \ \text{trimCharacters} \ ])
\]

A \text{CharacterExpression} is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string, or any Java data type that directly maps to a built-in type that is implicitly converted to a string.

RTRIM returns NULL if either \text{CharacterExpression} or \text{trimCharacters} evaluates to null.

If \text{trimCharacters} is not specified, it defaults to ‘ ’ (blank). It can be multiple characters.

**Examples**

```sql
-- returns ‘ asdf’
VALUES
RTRIM(’ asdf ‘)

-- returns ‘asdf’
VALUES
RTRIM(’asdf’, ’sa’)

-- returns ‘as’
VALUES
RTRIM(’asdf’, ’fd’)

-- returns ‘asdfg’
VALUES
RTRIM(’asdgffffff’, ’df’)
```
RUNTIMESTATISTICS()

RUNTIMESTATISTICS() returns an object that implements the COM.cloudscape.types.RunTimeStatistics interface. A RunTimeStatistics object contains information about the query that just ran—compilation and execution time and a statement execution plan. The information it provides helps you evaluate the way Cloudscape is executing complex queries and may help you determine whether an index is being used as you expected and whether you can rewrite the query to run faster.

Statistics must be on for this function to return an object; it returns an object describing the most recently completed query associated with this connection. For more information about when the object is returned, see “SET RUNTIMESTATISTICS statement” on page 1-111. If the statistics timing attribute is also on, the object returns timing information as well.

NEW: RunTimeStatistics objects are serializable, so you can store them in a table for analysis later. The format and content of these objects typically change from release to release. Beginning in Version 3.0, there will be an upgrade path for stored RunTimeStatistics objects. There is no upgrade path from previous versions to Version 3.0.

Syntax

RUNTIMESTATISTICS()

Example

This example uses ij.

```
-- set ij's maximum display width to very large so
-- you can see the full text
ij >maximumdisplaywidth 5000;
-- turn on runtimestatistics and statistics timing
ij> SET RUNTIMESTATISTICS ON;
ij> SET STATISTICS TIMING ON;
-- issue a query
ij> SELECT COUNT(*)
FROM sys.syscolumns;
-- dump out the run time statistics info
-- for the previous query
ij> VALUES RunTimeStatistics().toString();
Parse Time: 1011
Bind Time: 1492
```
How to Access and Use the Information

**NOTE:** The easiest way to view the information returned is to use Cloudview. For information, see “Viewing Runtime Statistics in Cloudview” on page 3-24 in *Tuning Cloudscape*.

To access the information in the object returned, access one or more of the object’s method calls from within an SQL-J statement. For information about the methods, see the API for the `COM.cloudscape.types.RuntimeStatistics` interface (located in *Cloudscape Reference Manual*).
Executing the `toString` method provides all the information in the object:

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

Executing the `getScanStatisticsText` method provides information about only those nodes that access a table or index. Using this method instead of the more complete `toString` method helps you weed out information that is not useful, if all you want to know is the access path:

```java
VALUES RUNTIMESTATISTICS().getScanStatisticsText()
```

For an in-depth discussion of how to analyze the information, see “Working with RunTimeStatistics” on page 3-18 in Tuning Cloudscape.

### Information Presented

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 when the statement is prepared. Compilation is composed of four 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 “0” time for each stage.

The `RunTimeStatistics` object also provides information about the statement execution plan. A statement execution plan is composed of a tree of `ResultSet` nodes. A `ResultSet` 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 inserts, updates, and deletes, rows flow out the top, where they are inserted, updated, or deleted. For selects (queries), rows flow out the top into a `ResultSet` that is returned to the user.

Nodes at the bottom are executed before the nodes above them. For example, a statement might show an `IndexScan` at the bottom (looking up key values in the index), then an `IndexRowToBaseRowResultSet` above that (retrieving the corresponding rows), then a `SortResultSet` on top (sorting the results).

Table 1-4 shows the many possible `ResultSet` nodes that might appear in an execution plan.
To learn more about how Cloudscape executes statements, see Chapter 4, “DML Statements and Performance” and Appendix A, “Internal Language Transformations” in *Tuning Cloudscape*.

### Table 1-4 Types of ResultSet Nodes in a DML Execution Plan

<table>
<thead>
<tr>
<th>ResultSet Node Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnyResultSet</td>
<td>Appears at the top of a quantified predicate subquery (EXISTS, IN, ANY, ALL). Calls a child node to evaluate the subquery and returns a row that contains a column with a boolean value that denotes whether the subquery evaluated to true.</td>
</tr>
<tr>
<td>CurrentOfResultSet</td>
<td>Appears under a DeleteNode or UpdateNode in the execution tree for a positioned delete or update. It returns the current row, if one exists, from the specified cursor.</td>
</tr>
<tr>
<td>DeleteResultSet</td>
<td>Appears at the top of the tree for a DELETE statement. All rows in this set are deleted from the base table and any indexes.</td>
</tr>
</tbody>
</table>
| DistinctScalarAggregateResultSet    | Represents nongrouped aggregation when there are one or more distinct aggregates. (Only one DISTINCT aggregate per query block is allowed.) A sort eliminates the duplicates when doing this type of aggregation. Does the aggregation for the following query: 
   ```sql
   select COUNT(c1), sum(distinct c2)
   from t1
   ``` |
| GroupedAggregateResultSet           | Represents grouped aggregation. Does the aggregation for the following query: 
   ```sql
   select c1, sum(c1) from t1 GROUP BY c1
   ``` |
| HashJoinExistsJoinResultSet         | Implements a hash join for an EXISTS type of join (one in which the right side of the join is probed only once per outer row).                |
| HashJoinLeftOuterJoinResultSet      | Implements a hash join for a left outer join. (Right outer joins are internally transformed into left outer joins during compilation.) Applies the outer join clause to the joined row to see if it qualifies. Returns those rows that qualify. |
| HashJoinResultSet                   | Implements a hash join between two tables. Returns rows from the join.                                                                      |
Table 1-4  Types of ResultSet Nodes in a DML Execution Plan (continued)

<table>
<thead>
<tr>
<th>ResultSet Node Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashScanResultSet</td>
<td>Implements a hash scan (the inner table or virtual table) of a hash join. Displays the key columns.</td>
</tr>
<tr>
<td>IndexRowToBaseRowResultSet</td>
<td>Gets a row id from an index key in an index scan (IndexScanResultSet) and returns the matching row from the base table.</td>
</tr>
<tr>
<td>IndexScanResultSet</td>
<td>Represents a scan on an index. Can have qualifiers, which are the predicates that are evaluated in the store level before the IndexScanResultSet gets them.</td>
</tr>
<tr>
<td></td>
<td>• An index scan with start and stop conditions is called a matching index scan; the start condition positions the scan at a starting point in the leaf level of the index; the scan traverses the leaf level of the index until the stop condition evaluates to false.</td>
</tr>
<tr>
<td></td>
<td>• An index scan without start and stop conditions is called a non-matching index scan; the scan traverses the entire leaf level of the index.</td>
</tr>
<tr>
<td></td>
<td>For more information about start and stop conditions and qualifiers for index traversal, see “What’s Optimizable?” on page 4-5 in Tuning Cloudscape.</td>
</tr>
<tr>
<td></td>
<td>For both types of index scans, qualifiers restrict the rows returned from the scan.</td>
</tr>
<tr>
<td>InsertResultSet</td>
<td>Appears at the top of the tree for an INSERT statement. All rows in this set are inserted into the target table and any indexes.</td>
</tr>
<tr>
<td>MaterializeResultSet</td>
<td>Represents the materialization of an ExternalVirtualTable into a temp table.</td>
</tr>
<tr>
<td>NestedLoopExistsJoinResultSet</td>
<td>Implements a nested loop join for an EXISTS type of join (one in which the right side of the join is probed only once per outer row).</td>
</tr>
<tr>
<td>NestedLoopJoinResultSet</td>
<td>Implements a nested loop join between two tables. Returns rows from the join.</td>
</tr>
<tr>
<td>NestedLoopLeftOuterJoinResultSet</td>
<td>Implements a nested loop join for a left outer join. (Right outer joins are internally transformed into left outer joins during compilation.) Applies the outer join clause to the joined row to see if it qualifies. Returns those rows that qualify.</td>
</tr>
</tbody>
</table>
### Table 1-4 Types of ResultSet Nodes in a DML Execution Plan (continued)

<table>
<thead>
<tr>
<th>ResultSet Node Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NormalizeResultSet</em></td>
<td>Does any necessary type conversions. Converts values in source row to the types that the parent ResultSet expects, and returns that row.</td>
</tr>
<tr>
<td><em>OnceResultSet</em></td>
<td>Appears at the top of an expression subquery (=, &lt;&gt;, &gt;, &gt;=, &lt;, &lt;= compared to a scalar subquery). Returns a row with the matching value from the subquery, if a single matching row exists, or a NULL if no matching value exists. The node throws an exception if a cardinality violation occurs (if more than one row is returned by the subquery).</td>
</tr>
<tr>
<td><em>ProjectRestrictResultSet</em></td>
<td>Can apply high-level restrictions and projections; evaluates those restrictions that are too expensive to push down to the base table or index (e.g., method calls and subqueries). Evaluates projections such as column elimination and expression evaluation for result columns. It also evaluates those restrictions that cannot be evaluated in a single table, such as t1.c1 = t2.c2. Includes two boolean fields, restriction and projection.</td>
</tr>
<tr>
<td><em>RowResultSet</em></td>
<td>Represents a row constructor and returns a single row. It has no children. For example, appears at the bottom of the tree for the statement “VALUES (1,2,3)”.</td>
</tr>
</tbody>
</table>
| *ScalarAggregateResultSet* | Represents nongrouped aggregation when there are no distinct aggregates. No sort is involved when doing this type of aggregation. Does the aggregation for the following query:  

  ```sql
  SELECT COUNT(c1), SUM(c2) FROM t1
  ```

| *SortResultSet*             | Represents a sort or an aggregation. Returns the result rows from the sort or aggregation.  

  *SortResultSet* represents sorts for ordering or duplication elimination. Aggregation has been broken out into new *ResultSet*. (See “ScalarAggregateResultSet” on page 1-166, “DistinctScalarAggregateResultSet” on page 1-164, and “GroupedAggregateResultSet” on page 1-164.) A boolean field, eliminateDuplicates indicates whether the *SortResultSet* eliminates duplicates. |
Subset of Information Presented

The portions of the statement plan that are returned in the `getScanStatisticsText` method are:

- `IndexRowToBaseRowResultSet`
- `TableScanResultSet`
- `IndexScanResultSet`

The output also contains information denoting the beginning and ending of a subquery block. (All nodes within the demarcation belong to a subquery.)

`IndexScanResultSets` and `TableScanResultSets` are the leaf nodes in the statement execution plan tree. The order in which the `IndexScanResultSets` and `TableScanResultSets` appear in the output String generally reflects their positions within the join order of the query block in which they appear.

The only current exception is for queries with UNIONs. There is no demarcation in the output between the two sides of a UNION. A UNION can appear within a derived table or in a subquery in the FROM list.

Multi-Threaded Applications

`RUNTIMESTATISTICS()` may not work as you would expect in an environment where threads are sharing a connection. `RUNTIMESTATISTICS()` returns the information from the most recently completed statement; if multiple threads are executing statements, you don’t really know which one executed last. The solution is to put the execution of the statement you want to evaluate and the call to

Cloudscape Version 3.0
RUNTIME STATISTICS() within a Java sync block and to similarly limit all other JDBC calls.

**Non-DML Statements**

The information returned for DDL statements may be sketchy. RUNTIME STATISTICS() is most meaningful for DML statements.

The ResultSet that can make up a query execution plan are subject to change.

The information contained in the String returned by getQueryPlan, toString, or getScanStatisticsText is subject to change.
SESSION_USER

SESSION_USER returns the authorization identifier or name of the current user. If there is no current user, it returns APP.

USER, CURRENT_USER, and SESSION_USER are synonyms.

Syntax

SESSION_USER

Example

VALUES SESSION_USER
The SUBSTRING function acts on a character string expression or a bit string expression. The type of the result is a VARCHAR in the first case and BIT VARYING in the second case.

The maximum length of the result is the maximum length of the source expression.

Syntax

```
SUBSTRING( { CharacterExpression | BitExpression } 
             FROM startPosition [ FOR lengthOfString ] )
```

`startPosition` and the optional `lengthOfString` are both integer expressions. (The first character or bit has a `startPosition` of 1; if you specify 0, Cloudscape assumes that you mean 1.) An exception is thrown if `lengthOfString` is specified and it is negative.

A `CharacterExpression` is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string (except a bit expression), or any Java data type that directly maps to a built-in type that is implicitly converted to a string.

For character expressions, both `startPosition` and `lengthOfString` refer to characters. For bit expressions, both `startPosition` and `lengthOfString` refer to bits.

If `lengthOfString` is not specified, SUBSTRING returns the substring of the source from the start position to the end of the source string. If `lengthOfString` is specified, SUBSTRING returns a VARCHAR or BIT VARYING with a length of `lengthOfString` starting at the `startPosition`.

Odd Cases

The function returns NULL if the source expression or `startPosition` evaluates to null or if `lengthOfString` is specified and it evaluates to null.

If `startPosition` evaluates to an integer that is greater than the length of the source, an empty character or bit expression is returned.

`startPosition` is allowed to be negative. If it is negative and `startPosition + lengthOfString` <= 0, an empty VARCHAR or VARBIT is returned. If `startPosition` is negative and `startPosition + lengthOfString` > 0, a character or bit expression of length `startPosition + lengthOfString` starting at position 1 is returned.

**NOTE:** SUBSTRING is very similar to SUBSTR. The different syntaxes are provided to allow compatibility with applications that are programmed for different vendors’ syntaxes. The only functional difference between them.
is that they have different semantics for a negative startPosition and a negative lengthOfString.

**Examples**

```
-- returns 34
VALUES SUBSTRING('12345' FROM 3 FOR 2)

-- returns 34
VALUES SUBSTRING(12345.toString() FROM 3 FOR 2)

-- returns 345
VALUES SUBSTRING('12345' FROM 3)

-- returns '12' --0 is converted to 1
VALUES SUBSTRING('12345' FROM 0 FOR 2)

-- returns 1
VALUES SUBSTRING('12345' FROM -2 FOR 3)

-- exception is thrown
VALUES SUBSTRING('12345' FROM 3 FOR -1)

-- returns null
SELECT SUBSTRING(stringColumn FROM 1 FOR 3)
FROM mytable
WHERE stringColumn IS NULL
```
The SUBSTR function acts on a character string expression or a bit string expression. The type of the result is a VARCHAR in the first case and BIT VARYING in the second case. The length of the result is the maximum length of the source type.

**Syntax**

```
SUBSTR({ CharacterExpression | BitExpression }, startPosition [, lengthOfString ] )
```

`startPosition` and the optional `lengthOfString` are both integer expressions. (The first character or bit has a `startPosition` of 1; if you specify 0, Cloudscape assumes that you mean 1.)

A `characterExpression` is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string (except a bit expression), or any Java data type that directly maps to a built-in type that is implicitly converted to a string.

For character expressions, both `startPosition` and `lengthOfString` refer to characters. For bit expressions, both `startPosition` and `lengthOfString` refer to bits.

SUBSTR returns NULL if `lengthOfString` is specified and it is less than zero.

If `startPosition` is positive, it refers to position from the start of the source expression (counting the first character as 1). If `startPosition` is negative, it is the position from the end of the source.

If `lengthOfString` is not specified, SUBSTR returns the substring of the expression from the `startPosition` to the end of the source expression. If `lengthOfString` is specified, SUBSTR returns a VARCHAR or VARBIT of length `lengthOfString` starting at the `startPosition`.

**Odd Cases**

The function returns NULL if the source expression or `startPosition` evaluates to null or if `lengthOfString` is specified and it evaluates to null.

If the `startPosition` evaluates to an integer that is greater than the length of the source SUBSTR returns an empty value of the appropriate type.

It is conceivable that `startPosition` is negative and its absolute value is >= the length of the source expression. In this case, if `lengthOfString` is not specified, the entire source expression is returned. If `lengthOfString` is specified and `startPosition + lengthOfString` <= 0, an empty value of the appropriate type is returned. If
lengthOfString is specified and startPosition + lengthOfString > 0, a value of length startPosition + lengthOfString starting at position 0 of the appropriate type is returned.

**NOTE:** SUBSTRING is very similar to SUBSTR. The different syntaxes are provided to allow compatibility with applications that are programmed for different vendors’ syntaxes. The only functional difference between them is that they have different semantics for a negative startPosition and a negative lengthOfString.

**Examples**

```
-- returns 34
VALUES SUBSTR('12345', 3, 2)

-- returns 34
VALUES SUBSTR(12345..toString(), 3, 2)

-- returns 345
VALUES SUBSTR('12345', 3)

-- returns '12' --0 is converted to 1
VALUES SUBSTR('12345', 0, 2)

-- returns 45
VALUES SUBSTR('12345', -2, 3)

-- returns null
VALUES SUBSTR('12345', -2, 3)

-- returns null
VALUES SUBSTR('12345', -2, 3)

SELECT SUBSTR(stringColumn, 1, 3)
FROM mytable
WHERE stringColumn IS NULL
```
**SUM**

SUM is an aggregate expression that evaluates the sum of the expression over a set of rows (see “Aggregates (Set Functions)” on page 1-129). SUM is allowed only on expressions that evaluate to numeric data types.

**Syntax**

```
SUM ( [ DISTINCT | ALL ] Expression )
```

The DISTINCT and ALL qualifiers eliminate or retain duplicates, but these qualifiers have no effect in a SUM expression. ALL is assumed if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1, 1, 1, 1, and 2, `SUM(col)` returns a greater value than `COUNT(DISTINCT col)`.

Only one DISTINCT aggregate expression per `SelectExpression` is allowed. For example, the following query is not allowed:

```
SELECT AVG (DISTINCT flying_time), SUM (DISTINCT miles)
FROM Flights
```

The `Expression` can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in numeric data type or to a Java data type that is automatically mapped to a built-in numeric data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a `java.lang.Integer` or `int` evaluates to an INTEGER.) If an expression evaluates to NULL, the aggregate skips that value. The resulting data type is the same as the expression on which it operates (it may overflow).

**Example**

```
-- find all the rooms booked
SELECT SUM (rooms_taken)
FROM HotelAvailability

-- use SUM on a method invocation
SELECT SUM (fixed_rate.doubleValue())
FROM FlightBookings

-- use SUM on multiple column references
-- (find the total value of all seats purchased)
SELECT SUM(fixed_rate.doubleValue() * number_seats)
FROM FlightBookings
```
The TRIM function trims one or more instances of a single character from the beginning or end (or both) of a character expression. By default, TRIM trims one blank space from the beginning and one from the end of a character expression.

The type of the result is VARCHAR. The length of the result is the maximum length of the source type.

**Syntax**

```
TRIM( [ [ { LEADING | TRAILING | BOTH } ]
   [ trimCharacter ] FROM ] CharacterExpression )
```

A `CharacterExpression` is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string, or any Java data type that directly maps to a built-in type that is implicitly converted to a string.

`BOTH` is implicit if `LEADING`, `TRAILING`, or `BOTH` is not specified.

`trimCharacter` must be one single character. If `trimCharacter` is not specified, `' '` (blank) is implicit.

If `TRAILING` is specified, TRIM removes one or more trailing characters equal to `trimCharacter` from the result.

If `LEADING` is specified, TRIM removes one or more leading characters equal to `trimCharacter` from the result.

If `BOTH` is specified (or implied by its absence), TRIM removes one or more leading or trailing characters equal to the `trimCharacter` from the result.

TRIM(`CharacterExpression`) is the same as TRIM(`BOTH ' ' FROM CharacterExpression`).

If `FROM` is specified, you must specify either or both of the following:

- `LEADING`, `TRAILING`, or `BOTH`
- `trimCharacter`

If either `trimCharacter` or `CharacterExpression` is a user-specified null, the result of the expression is null.

**Examples**

```sql
-- returns 'asd f'
VALUES
TRIM(' asd f ')
```
-- returns 'asdf'
VALUES
TRIM('x' FROM 'xasdf x')

-- throws exception,
VALUES
TRIM(TRAILING 'as' FROM 'asdf')
-- trimCharacter is not a single character

-- returns 'asd'
VALUES
TRIM(TRAILING 'f' FROM 'asdf')

-- returns 'babaac'
VALUES
TRIM ('a' FROM 'aababaacaa')
UPPER

UPPER takes a character expression as a parameter and returns a string in which all alpha characters have been converted to uppercase.

Syntax

```
UPPER ( CharacterExpression )
```

A `characterExpression` is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string, or any Java data type that directly maps to a built-in type that is implicitly converted to a string.

If the parameter type is CHAR or LONG VARCHAR, the return type is CHAR or LONG VARCHAR. Otherwise, the return type is VARCHAR.

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

Example

```
-- returns 'ASD1#W'
UPPER('aSD1#w')
```
USER

USER returns the authorization identifier or name of the current user. If there is no current user, it returns `APP`.

USER, CURRENT_USER, and SESSION_USER are synonyms.

Syntax

```
USER
```

Example

```
VALUES USER
```
User-Defined Aggregate

Cloudscape allows you to create your own aggregate functions. After you create the Java classes that implement the aggregation and create the aggregate with the CREATE AGGREGATE statement, you can use those aggregate from within SQL-J the same way you use the built-in aggregates.

The resulting data type of the user-defined aggregate function is defined by the implementing class. If you provide the correct implementation, a user-defined aggregate can return any built-in type or any user-defined Java data type.

For information about creating the Java classes that implement the aggregation, see “Programming User-Defined Aggregates” on page 5-27 in the Cloudscape Developer’s Guide.

Syntax

\[ \text{AggregateName} \left( \begin{array}{c}
\text{DISTINCT} \mid \text{ALL}
\end{array} \right) \text{Expression} \]

Examples

-- MAXBUTONE is a user-defined aggregate
SELECT MAXBUTONE(DISTINCT miles)
FROM Flights

SELECT STDEV(flying_time)
FROM Flights

NEW: User-defined aggregates are new in Version 3.0.
Data Types

- “Built-In Type Overview” on page 1-180
- “SQL-J Types, Java Types, and JDBC Types” on page 1-181
- “Numeric Types” on page 1-182
- “Implicit Type Conversion of String Types” on page 1-186
- “Comparing Booleans with Numeric Types and Assigning Numeric Types to Booleans” on page 1-188
- “Data Types and Comparison, Sorting, and Ordering” on page 1-189

Built-In Type Overview

The SQL-J type system is used by the language compiler to determine the compile-time type of an expression and by the language execution system to determine the runtime type of an expression, which can be a subtype or implementation of the compile-time type.

Each type has associated with it values of that type. In addition, values in the database or resulting from expressions can be NULL, which means the value is missing or unknown. Although there are some places where the keyword NULL can be explicitly used, it is not in itself a value, because it needs to have a type associated with it.

The syntax presented in this section is the syntax you use when specifying a column’s data type in a CREATE TABLE statement.

Each built-in type in SQL-J has a Java class associated with it. For more information, see “Column Values and Type Correspondence” on page 1-243.

Cloudscape Version 3.0 supports the following data types:

- “BIT” on page 1-190
- “BOOLEAN” on page 1-193
- “CHAR” on page 1-194
- “DATE” on page 1-195
- “DECIMAL” on page 1-196
- “DOUBLE PRECISION” on page 1-198
- “FLOAT” on page 1-199
- “INTEGER” on page 1-200
Cloudscape also supports the use of Java data types, and it supplies a number of built-in Java data types that are used internally or in system tables. See “Java Data Types (User-Defined Data Types)” on page 1-211.

### SQL-J Types, Java Types, and JDBC Types

Built-in types have corresponding Java data types at compile time. This correspondence is pertinent at compile time, when Cloudscape validates data types for inserts, joins, unions, and any statement in which two values must be compared somehow. A `PreparedStatement` is compiled when it is prepared; a `Statement` is compiled when it is executed. Values associated with a column of the built-in type automatically map, when an SQL-J statement is compiled, to values of the corresponding Java data type during compilation. For example, an insert into a column of type INTEGER expects a value of an INTEGER data type. Because the INTEGER data type has a compile-time corresponding Java data type of `java.lang.Integer`, Cloudscape also accepts a value of `java.lang.Integer` at compile time.

```sql
-- the compiler determines that the data type of the
-- value to be inserted into myIntColumn matches
-- the expected type
INSERT INTO myTable (myIntColumn)
VALUES (new java.lang.Integer('2'))
```

Since built-in types do not correspond to Java primitives at compile time, you cannot construct a Java primitive with a simple literal within the context of an SQL-J statement. For example, the literal 1 is an INTEGER value (which corresponds to

Cloudscape Version 3.0
a `java.lang.Integer`). This makes calling Java methods that take primitives a bit tricky; see “SQL-J to Java Type Correspondence” on page 1-244.

However, when working in a Java program using JDBC methods to set the runtime value of dynamic parameters or to retrieve values from a `ResultSet`, you typically do work with Java primitives. For example, you would typically use the `stint` method of `java.sql.PreparedStatement` to insert values into an INTEGER column and the `getInt` method of `java.sql.ResultSet` to retrieve values from that column. That’s because of the type correspondence set up by JDBC. A JDBC INTEGER type automatically maps to a Java `int` type; an SQL-J INTEGER type automatically maps to a JDBC INTEGER type. So, within the context of a JDBC program, type correspondence works differently than within the compile-time context of a SQL-J statement.

The data type man pages in this section list the corresponding compile-time Java type for the data type. (For more detailed reference on the compile-time correspondence between SQL-J and Java data types, see “SQL-J to Java Type Correspondence” on page 1-244.) They also list the corresponding JDBC type (but not its corresponding Java type). For information on how JDBC types map to Java types, see the JDBC documentation.

**Numeric Types**

- “Numeric Type Overview” on page 1-182
- “Numeric Type Promotion in Expressions” on page 1-183
- “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 1-184
- “Scale for Decimal Arithmetic” on page 1-185

**Numeric Type Overview**

Numeric types include the following types, which provide storage of varying sizes:

- Integer numerics
  - TINYINT (1 byte)
  - SMALLINT (2 bytes)
  - INTEGER (4 bytes)
  - LONGINT (8 bytes)
- Approximate or floating-point numerics
Data Types

- REAL (4 bytes)
- DOUBLE PRECISION (8 bytes)
- FLOAT (an alias for DOUBLE PRECISION or REAL)
- Exact numeric
  - DECIMAL (storage based on precision)
  - NUMERIC (an alias for DECIMAL)

**Numeric Type Promotion in Expressions**

In expressions that use only integer types, Cloudscape promotes the type of the result to at least INTEGER. In expressions that mix integer with non-integer types, Cloudscape promotes the result of the expression to the highest type in the expression. Table 1-5 shows the promotion of data types in expressions.

<table>
<thead>
<tr>
<th>Largest Type That Appears in Expression</th>
<th>Resulting Type of Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>REAL</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>LONGINT</td>
<td>LONGINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TINYINT</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

For example:

```sql
-- returns a double precision
VALUES 1 + 1.0e0

-- returns a decimal
VALUES 1 + 1.0

-- returns an integer
VALUES CAST (1 AS TINYINT) + CAST (1 AS TINYINT)
```
Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type

An attempt to put a floating-point type of a larger storage size into a location of a smaller size fails only if the value cannot be stored in the smaller-size location. For example:

```sql
create table mytable (r REAL, d DOUBLE PRECISION);
```

0 rows inserted/updated/deleted

```sql
INSERT INTO mytable (r, d) values (3.4028236E38, 3.4028235E38);
```

ERROR X0X41: The number ‘3.4028236E38’ is outside the range of the real datatype.

You can store a floating point type in an INTEGER column; the fractional part of the number is truncated. For example:

```sql
INSERT INTO mytable(integer_column) values (1.09e0);
```

1 row inserted/updated/deleted

```sql
SELECT integer_column
FROM mytable;
```

```
1
```

Integer types can always be placed successfully in approximate numeric values, although with the possible loss of some precision.

Integers can be stored in decimals if the DECIMAL precision is large enough for the value. For example:

```sql
insert into mytable (decimal_column)
VALUES (55555555556666666666);
```

ERROR X0Y21: The number ‘55555555556666666666’ is outside the range of the target DECIMAL/NUMERIC(5,2) datatype.

An attempt to put an integer value of a larger storage size into a location of a smaller size fails if the value cannot be stored in the smaller-size location. For example:

```sql
INSERT INTO mytable (tinyint_column) values 6555;
```

ERROR X0X76: The number ‘6555’ is outside the range of the tinyint datatype.

**NOTE:** When truncating trailing digits from a NUMERIC value, Cloudscape rounds down.
**Scale for Decimal Arithmetic**

SQL-J statements can involve arithmetic expressions that use decimal data types of different *precisions* (the total number of digits, both to the left and to the right of the decimal point) and *scales* (the number of digits of the fractional component). The precision and scale of the resulting decimal type depend on the precision and scale of the operands.

Given an arithmetic expression that involves two decimal operands:

- \( lp \) stands for the precision of the left operand
- \( rp \) stands for the precision of the right operand
- \( ls \) stands for the scale of the left operand
- \( rs \) stands for the scale of the right operand

Use the following formulas to determine the scale of the resulting data type for the following kinds of arithmetical expressions:

- **multiplication**
  \( ls + rs \)

- **division**
  \( \max(ls + rp - rs + 1, 4) \)

- **AVG()**
  \( \max(\max(ls, rs), 4) \)

- **all others**
  \( \max(ls, rs) \)

For example, the scale of the resulting data type of the following expression is 7:

\[
11.0/1111.33
\]

// \( \max((2 + 6 − 2 + 1), 4) \)

Use the following formulas to determine the precision of the resulting data type for the following kinds of arithmetical expressions:

- **multiplication**
  \( lp + rp \)

- **addition**
  \( 2 * (p - s) + s \)

- **division**
  \( lp - ls + rp + \max(ls + rp - rs + 1, 4) \)

- **all others**
  \( \max(lp - ls, rp - rs) + 1 + \max(ls, rs) \)
Implicit Type Conversion of String Types

Cloudscape supports a number of *implicit* conversions between string and non-string types.

String types consist of:

- CHAR (string literals are always of type CHAR)
- LONG VARCHAR
- VARCHAR

Implicit Conversions from Strings to Other Built-In Data Types

- **Comparison to non-string type**
  
  If a comparison is made between a non-string, built-in type and a string type, the string type is converted to the non-string, built-in type, as if the string type were explicitly cast to the type of the non-string, built-in type.
  
  For example, the following expressions are treated the same:

  ```
  1 < '2'
  1 < CAST ('2' AS INT)
  WHERE 3.5 = '3.5'
  WHERE 3.5 = CAST ('3.5' AS DECIMAL)
  
  -- first the INTEGER literals are implicitly cast to strings
  23 < '2' || '2'
  23 < CAST ('2' || '2' AS INT)
  ```

  **NOTE:** In a statement in which there is an equijoin condition that requires an *implicit* (var)char conversion, the statement is disqualified for consideration for a hash join.

  A cast to the type of the non-string type may not be what you want, however. For example, the following comparisons raise an exception, because Cloudscape attempts to cast '2.5' to an integer, which contains an illegal character for the integer data type:

  ```
  1 < '2.5'
  1 < CAST ('2.5' AS INT)
  ```

- **Use in function or assignment expecting non-string type**

  If a string type is used in a non-string type built-in expression as an argument to a function expecting a non-string type built-in argument, the string type is converted to the expected non-string type built-in type before use. *Function* here means arithmetic operators and built-in functions as
well as assignments (as in INSERT and UPDATE) and unions. (It does not mean a Java method or method alias; no implicit conversion is done on the arguments to Java methods.)

For an arithmetic expression, at least one operand of an arithmetic operator must be a numeric type. (An exception is thrown if both operands are string types.) The string type is implicitly cast to the same numeric type as the other operand.

For example, the following expressions are treated the same:

```
INSERT INTO T (intColumn) VALUES '1'
INSERT INTO T (intColumn) VALUES (CAST ('1' AS INT))
VALUES 7.2 + '5.0'
VALUES 7.2 + (CAST '5.0' AS DECIMAL)
```

The following expression raises an exception:

```
VALUES '3' + '5'
```

The limitations on implicit conversions when compared to explicit conversions are:

- SUM and AVG aggregates are not supported on string types.
- If a string type is passed to the EXTRACT() built-in function, the string type is implicitly cast to a DATE when extracting the YEAR, MONTH, or DAY and implicitly cast to a TIME when extracting the HOUR, MINUTE, or SECOND. When using an explicit CAST, you can cast the string to the desired data type.

For the rules on invalid characters and when leading and trailing spaces are stripped out, see “CAST and Implicit Conversions from Character Strings” on page 1-138.

### Implicit Conversion to Strings from Other Built-In Data Types

- **Use in Function or Assignment Expecting String Type**
  
  If a non-string built-in type is used in a string expression or as an argument to a string function, the non-string built-in type is converted to a string type before use. *Function* here means string operators ("||"), the LIKE predicate, and built-in functions as well as assignments (as in INSERT and UPDATE). For example, the following expressions are treated the same:

```
INSERT INTO T (charColumn) VALUES 1
INSERT INTO T (charColumn) VALUES (CAST (1 AS CHAR(1)))
```
For the rules on size compatibility and padding with spaces, see “Size Requirements for Explicit and Implicit Conversions to Character Strings” on page 1-137.

**Limitations on Implicit Conversions from or to String Types**

The cases in which an explicit CAST is required to or from a string type are:

- From Java data types to string types. You can call the `toString` method if you want this behavior.
- For the receiver or arguments to a Java method call.

**Comparing Booleans with Numeric Types and Assigning Numeric Types to Booleans**

Booleans can be compared with any numeric type, and you can assign a numeric type to a boolean and vice versa.

- When you compare a number with a boolean, zero equals false, and any other value equals true.
- When you assign a number to a boolean, zero becomes false, and any other value becomes true. For example, if you insert 0 into a boolean column, the 0 will be converted to false to be stored in the column, and if you insert –1, the value will be converted to true.
- When you assign a boolean to a number, false becomes zero, and true becomes 1. For example, if you insert false into an int column, the false is converted to 0 to be stored in the column, and if you insert true, the value will be converted to 1.
- You cannot pass booleans to methods expecting numbers, and vice versa.
## Data Types and Comparison, Sorting, and Ordering

### Table 1-6 Data Types and Sorting, Ordering, and Comparisons

<table>
<thead>
<tr>
<th></th>
<th>All Types</th>
<th>Java Data Types that Correctly Implement equals and hashCode</th>
<th>Orderable Java Data Types</th>
<th>Built-in Types (and Corresponding Java Data Types)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparisons = and &lt;&gt;</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Comparisons &gt;, &gt;=, &lt;, &lt;=</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Ordering operations include DISTINCT, GROUP BY, UNION, and ORDER BY</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
BIT

A BIT field allows you to store bit strings of a specified length. It is useful for unstructured data where character strings are not appropriate (e.g., images).

**Syntax for Column Definition**

BIT [ (length) ]

*length* is an unsigned integer literal designating the length in bits.

The default *length* for a BIT is 1.

**Corresponding Compile-Time Java Type**

byte[

**JDBC Metadata Type (java.sql.Types)**

BINARY

BIT stores fixed-length bit strings. If a BIT value is smaller than the target BIT, it is padded with zeros. If a BIT value is larger than the target length, a bit right truncation exception is raised.

Comparisons of BIT and BIT VARYING values are precise. For two bit strings to be equal, they must be exactly the same length. (This differs from the way some other DBMSs handle BINARY values but works as specified in SQL-92.)

An operation on a BIT VARYING and a BIT value (e.g., a concatenation) yields a BIT VARYING value.

**Literal Examples**

You can express bit literals in base-2 (binary) or base-16 (hexadecimal) format. Base-2 literals begin with *B*, and base-16 literals begin with an *X*.

The type of a bit literal is always a BIT.

```
-- length 8
VALUES X'1a'

-- length 4
VALUES B'1010'

-- binary bit literal with length 3
B'011'
```
When bit strings are converted to character strings, they are always represented in hexadecimal format.

```sql
CREATE TABLE t (b bit(8));
INSERT INTO t VALUES (B'11111111');
SELECT * FROM t;
```

-- yields the following output

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
</tr>
</tbody>
</table>

Bit literals specified in binary format are taken to be the length of the number of bits represented. Bit literals specified in hexadecimal format are padded out to the nearest 8 bits. Therefore, X’a’ is automatically converted to X’a0’.

**Implementation-Defined Aspects**

The only limit on the length of BIT and BIT VARYING data types is the constraint of the integer used to specify the length, the value `java.lang.Integer.MAX_VALUE`. Internally, BIT/BIT VARYING columns are stored in eight-bit chunks.
BIT VARYING

The BIT VARYING field allows you to store bit strings of a specified length. Like the BIT data type, it is useful for unstructured data where character strings are not appropriate (e.g., images).

**Syntax for Column Definition**

```plaintext
BIT VARYING (length)
```

`length` is an unsigned integer literal designating the length in bits.

Unlike the case for the BIT data type, there is no default `length` for a BIT VARYING type.

**Corresponding Compile-Time Java Type**

`byte[]`

**JDBC Metadata Type (java.sql.Types)**

`BINARY`

BIT VARYING stores variable-length bit strings. Unlike BIT values, BIT VARYING values are not padded out to the target length. If a BIT VARYING value is larger than the target length, a bit right truncation exception is raised.

Comparisons of BIT and BIT VARYING values are precise. For two bit strings to be equal, they must be `exactly` the same length. (This differs from the way some other DBMSs handle BINARY values but works as specified in SQL-92.)

An operation on a BIT VARYING and a BIT value (e.g., a concatenation) yields a BIT VARYING value.

**Literal Examples**

The type of a bit literal is always a BIT, not a BIT VARYING.

**Implementation-Defined Aspects**

The only limit on the length of BIT and BIT VARYING data types is the constraint of the integer used to specify the length, the value `java.lang.Integer.MAX_VALUE`. 
Syntax for Column Definition

**BOOLEAN**

Corresponding Compile-Time Java Type

`java.lang.Boolean`

JDBC Metadata Type (`java.sql.Types`)

**BIT**

Booleans have the value TRUE, FALSE, or NULL.

Booleans cannot be implicitly converted to other types, and other types cannot be
converted to booleans, except that a boolean can be inserted into a
`java.lang.Boolean` column (and vice versa) and that JDBC performs some
conversions for its `get` and `set` methods. CAST allows specific additional
conversions.

The value TRUE is considered to sort higher than the value FALSE, for
comparisons and ORDER BY.

Literal Examples

VALUES TRUE
VALUES FALSE
CHAR

CHAR provides for fixed-length storage of strings.

Syntax for Column Definition
CHAR [(length)]

length is an unsigned integer literal. The default length for a CHAR is 1.

Corresponding Compile-Time Java Type
java.lang.String

JDBC Metadata Type (java.sql.Types)
CHAR

Cloudscape inserts spaces to pad a string value shorter than the expected length. Cloudscape truncates spaces from a string value longer than the expected length. Characters other than spaces cause an exception to be raised. When binary comparison operators are applied to CHARs, the shorter string is padded with spaces to the length of the longer string.

When CHARs and VARCHARs are mixed in expressions, the shorter value is padded with spaces to the length of the longer value.

The type of a string literal is CHAR.

Implementation-Defined Aspects

The only limit on the length of CHAR data types is the value java.lang.Integer.MAX_VALUE.

Literal Examples

-- within a string literal use two single quotation marks
-- to represent a single quotation mark or apostrophe
VALUES ‘hello this is Joe’s string’
DATE

DATE provides for storage of a year-month-day in the range supported by `java.sql.Date`.

**Syntax for Column Definition**

```
DATE
```

**Corresponding Compile-Time Java Type**

`java.sql.Date`

**JDBC Metadata Type (java.sql.Types)**

```
DATE
```

Dates, times, and timestamps must not be mixed with one another in expressions. SQL-J is more permissive than SQL-92 in the values and formats it permits in date/time literals. Any value that is recognized by the `java.sql.Date`, `java.sql.Time` or `java.sql.Timestamp.valueOf(String)` method is permitted in a column of the corresponding SQL-J date/time data type.

**Literal Syntax**

```
[DATE] 'YYYY-MM-DD'
```

**Literal Examples**

VALUES DATE '1994-02-23'

VALUES '1993-09-01'
DECIMAL

DECIMAL provides an exact numeric in which the precision and scale can be arbitrarily sized. You can specify the precision (the total number of digits, both to the left and the right of the decimal point) and the scale (the number of digits of the fractional component). The amount of storage required is based on the precision.

Syntax for Column Definition

{ DECIMAL | DEC } [(precision [, scale ])]

The precision must be between 1 and java.lang.Integer.MAX_VALUE (2147483647). The scale must be less than or equal to the precision.

If the scale is not specified, the default scale is 0. If the precision is not specified, the default precision is 18.

An attempt to put a numeric value into a DECIMAL is allowed as long as any non-fractional precision is not lost. When truncating trailing digits from a DECIMAL value, Cloudscape rounds down.

For example:

-- this cast loses only fractional precision
values cast (1.798765 AS decimal(5,2));
SQLCol1
--------
1.79
-- this cast does not fit
values cast (1798765 AS decimal(5,2));
SQLCol1
--------
ERROR X0Y21: The number '1798765' is outside the range of the target DECIMAL/NUMERIC(5,2) datatype.

When mixed with other data types in expressions, the resulting data type follows the rules shown in “Numeric Type Promotion in Expressions” on page 1-183.

See also “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 1-184.

When two decimal values are mixed in an expression, the scale and precision of the resulting value follow the rules shown in “Scale for Decimal Arithmetic” on page 1-185.
Corresponding Compile-Time Java Type

java.math.BigDecimal

JDBC Metadata Type (java.sql.Types)

DECIMAL

Literal Examples

VALUES 123.456
VALUES 0.001

Integer literals too big for LONGINT are made DECIMAL literals.
DOUBLE PRECISION

The DOUBLE PRECISION data type provides 8-byte storage for numbers using IEEE floating-point notation.

Syntax for Column Definition

DOUBLE PRECISION

Minimum Value

4.9E-324 (java.lang.Double.MIN_VALUE)

Maximum Value

1.7976931348623157E308 (java.lang.Double.MAX_VALUE)

Corresponding Compile-Time Java Type

java.lang.Double

JDBC Metadata Type (java.sql.Types)

DOUBLE

When mixed with other data types in expressions, the resulting data type follows the rules shown in “Numeric Type Promotion in Expressions” on page 1-183.

See also “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 1-184.

Literal Examples

3421E+09
425.43E9
9E-10
4356267544.32333E+30
FLOAT

The FLOAT data type is an alias for a REAL or DOUBLE PRECISION data type, depending on the precision you specify.

Syntax for Column Definition

FLOAT [ (precision) ]

The default precision for FLOAT is 53 and is equivalent to DOUBLE PRECISION. A precision of 23 or less makes FLOAT equivalent to REAL. A precision of 24 or greater makes FLOAT equivalent to DOUBLE PRECISION. If you specify a precision of 0, you get an error. If you specify a negative precision, you get a syntax error.

Corresponding Compile-Time Java Type

java.lang.Float (precision of 23 or less) or java.lang.Double (precision of 24 or greater)

JDBC Metadata Type (java.sql.Types)

REAL or DOUBLE

Minimum Value

4.9E-324 (java.lang.Double.MIN_VALUE) if using a precision of 24 or greater
1.4E-45 (java.lang.Float.MIN_VALUE) if using a precision of 23 or less

Maximum Value

1.7976931348623157E308 (java.lang.Double.MAX_VALUE) if using a precision of 24 or greater.
3.4028235E38 (java.lang.Float.MAX_VALUE) if using a precision of 23 or less

When mixed with other data types in expressions, the resulting data type follows the rules shown in “Numeric Type Promotion in Expressions” on page 1-183.

See also “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 1-184.
INTEGER

INTEGER provides 4 bytes of storage for integer values.

Syntax for Column Definition

{ INTEGER | INT }

Corresponding Compile-Time Java Type

java.lang.Integer

JDBC Metadata Type (java.sql.Types)

INTEGER

Minimum Value

-2147483648 (java.lang.Integer.MIN_VALUE)

Maximum Value

2147483647 (java.lang.Integer.MAX_VALUE)

When mixed with other data types in expressions, the resulting data type follows the rules shown in “Numeric Type Promotion in Expressions” on page 1-183.

See also “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 1-184.

Literal Examples

3453
425
LONG VARBINARY, LONG BIT VARYING, LONG BINARY

The LONG VARBINARY type allows storage of bit strings of unlimited length. It is identical to BIT, except that you do not have to specify a maximum length when creating columns of this type.

Syntax for Column Definition

```
{ LONG VARBINARY | LONG BIT VARYING | LONG BINARY }
```

Corresponding Compile-Time Java Type

byte[

JDBC Metadata Type (java.sql.Types)

LONGVARBINARY

When you are converting from Java values to SQL values, no Java type corresponds to LONG VARBINARY.
LONG VARCHAR

The LONG VARCHAR type allows storage of bit strings of unlimited length. It is identical to VARCHAR, except that you do not have to specify a maximum length when creating columns of this type.

Syntax for Column Definition

```
LONG VARCHAR
```

Corresponding Compile-Time Java Type

```
java.lang.String
```

JDBC Metadata Type (java.sql.Types)

```
LONGVARCHAR
```

When you are converting from Java values to SQL values, no Java type corresponds to LONG VARCHAR.
LONGINT

LONGINT provides 8 bytes of storage for integer values.

Syntax for Column Definition
LONGINT

Corresponding Compile-Time Java Type
java.lang.Long

JDBC Metadata Type (java.sql.Types)
BIGINT

Minimum Value
-9223372036854775808 (java.lang.Long.MIN_VALUE)

Maximum Value
9223372036854775807 (java.lang.Long.MAX_VALUE)

When mixed with other data types in expressions, the resulting data type follows the rules shown in “Numeric Type Promotion in Expressions” on page 1-183.

An attempt to put an integer value of a larger storage size into a location of a smaller size fails if the value cannot be stored in the smaller-size location. Integer types can always successfully be placed in approximate numeric values, although with the possible loss of some precision. LONGINTs can be stored in DECIMALs if the DECIMAL precision is large enough for the value.

Literal Example
9223372036854775807
NUMERIC

NUMERIC is a synonym for DECIMAL and behaves the same way. See “DECIMAL” on page 1-196.

Syntax for Column Definition

NUMERIC [(precision [, scale ])]

Corresponding Compile-Time Java Type

java.math.BigDecimal

JDBC Metadata Type (java.sql.Types)

NUMERIC

Literal Examples

123.456
.001
The REAL data type provides 4 bytes of storage for numbers using IEEE floating-point notation.

Syntax for Column Definition

```
REAL
```

Corresponding Compile-Time Java Type

```
java.lang.Float
```

JDBC Metadata Type (java.sql.Types)

```
REAL
```

Minimum Value

```
1.4E-45 (java.lang.Float.MIN_VALUE)
```

Maximum Value

```
3.4028235E38 (java.lang.Float.MAX_VALUE)
```

When mixed with other data types in expressions, the resulting data type follows the rules shown in “Numeric Type Promotion in Expressions” on page 1-183.

See also “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 1-184.

Literals always map to DOUBLE PRECISION; use a CAST to convert a literal to a REAL.
SMALLINT

SMALLINT provides 2 bytes of storage.

Syntax for Column Definition

\texttt{SMALLINT}

Corresponding Compile-Time Java Type

\texttt{java.lang.Short}

JDBC Metadata Type (\texttt{java.sql.Types})

SMALLINT

Minimum Value

-32768 (\texttt{java.lang.Short.MIN\_VALUE})

Maximum Value

32767 (\texttt{java.lang.Short.MAX\_VALUE})

3.4028235E38 (\texttt{java.lang.Float.MAX\_VALUE})

When mixed with other data types in expressions, the resulting data type follows the rules shown in “Numeric Type Promotion in Expressions” on page 1-183.

See also “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 1-184.

Literals in the appropriate format always map to INTEGER or LONGINT, depending on their length; use a CAST to convert a literal to a SMALLINT.
**TIME**

TIME provides for storage of an hour-minutes-seconds-fractional-seconds value with six digits in the fractional-seconds part.

**Syntax for Column Definition**

```
TIME
```

**Corresponding Compile-Time Java Type**

```
java.sql.Time
```

**JDBC Metadata Type** (java.sql.Types)

```
TIME
```

Dates, times, and timestamps cannot be mixed with one another in expressions except with a CAST.

SQL-J is more permissive than SQL-92 in the values and formats it permits in date/time literals. Any value that is recognized by the `java.sql.Date`, `java.sql.Time`, or `java.sql.Timestamp valueOf(String)` method is permitted in a column of the corresponding SQL-J date/time data type.

**Literal Syntax**

```
TIME'HH:MM:SS[.FFFFFF']
```

**Literal Examples**

```
TIME'15:09:02'
'15:09:02'
```
TIMESTAMP

TIMESTAMP stores a combined DATE and TIME value to be stored. It permits a fractional-seconds value of up to nine digits.

Syntax for Column Definition

TIMESTAMP

Corresponding Compile-Time Java Type

java.sql.Timestamp

JDBC Metadata Type (java.sql.Types)

TIMESTAMP

Dates, times, and timestamps cannot be mixed with one another in expressions. SQL-J is more permissive than SQL-92 in the values and formats it permits in date/time literals. Any value that is recognized by the java.sql.Date, java.sql.Time or java.sql.Timestamp.valueOf(String) method is permitted in a column of the corresponding SQL-J date/time data type.

Literal Syntax

[TIMESTAMP]YYYY-MM-DD HH:MM:SS[.FFFFFFFF]

Literal Examples

'1960-01-01 23:03:20'
TIMESTAMP'1962-09-23 03:23:34.234'
TIMESTAMP'1960-01-01 23:03:20'
TINYINT

TINYINT provides 1 byte of storage for integer values.

Syntax for Column Definition

```
TINYINT
```

Corresponding Compile-Time Java Type

```
java.lang.Byte
```

JDBC Metadata Type (java.sql.Types)

TINYINT

Minimum Value

```
java.lang.Byte.MIN_VALUE (-128)
```

Maximum Value

```
java.lang.Byte.MAX_VALUE (127)
```

When mixed with other data types in expressions, the resulting data type follows the rules shown in “Numeric Type Promotion in Expressions” on page 1-183.

See also “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 1-184.

Literals of the appropriate format always map to INTEGER or LONGINT; use a CAST to convert a literal to a SMALLINT.
VARCHAR

VARCHAR provides for variable-length storage of strings.

**Syntax for Column Definition**

```
VARCHAR (length)
```

`length` is an unsigned integer literal, and it must not be greater than the constraint of the integer used to specify the length, the value `java.lang.Integer.MAX_VALUE`.

**Corresponding Compile-Time Java Type**

`java.lang.String`

**JDBC Metadata Type** (`java.sql.Types`)

`VARCHAR`

Cloudscape does not pad a VARCHAR value whose length is less than expected. Cloudscape truncates spaces from a string value when a length greater than the VARCHAR expected is provided. Characters other than spaces are not truncated, and instead cause an exception to be raised. When binary comparison operators are applied to VARCHARs, the lengths of the operands are not altered, and spaces at the end of the values are ignored.

When CHARs and VARCHARs are mixed in expressions, the shorter value is padded with spaces to the length of the longer value.

The type of a string literal is CHAR, not VARCHAR.
Java Data Types (User-Defined Data Types)

You store Java objects in columns using serialization. You define a column to hold a serialized form of Java class or interface, which allows objects of that Java type and any of its implementors or subtypes to be serialized into the column.

To store Java objects, you can insert a row of values using dynamic parameters and pass in the objects from the Java application, copy objects from other tables, or invoke methods that return Java objects.

For general information about dynamic parameters, see “Dynamic Parameters” on page 1-226. When using dynamic parameters with Java data types, use the setObject method of JDBC’s java.sql.PreparedStatement to set the values of the parameters.

Once objects are stored, you can apply method invocation or field access on the stored value of a column holding a Java class to access the methods or fields of the object it holds, respectively.

NEW: Beginning in Version 3.0, you can order by, group by, compare, and index on certain Java types. For more information, see “Orderable Java Data Types” on page 1-214

Syntax for Column Definition

SERIALIZE (JavaClassName | ClassAlias)

Inserting an object or directly updating the column serializes the object. Accessing the column deserializes the object.

Requirements for Serialization

The class or interface must:

- Implement the interface java.io.Serializable.
- Be declared public.
- Be available to the JVM in which it is running (i.e., be installed and accessible on the deployment machine’s class path at the time the column is created; see “Deploying Java Classes for Use as Java Data Types” on page 3-6 in the Cloudscape Developer’s Guide).

For detailed information on how to create serializable Java data types that you can store, see “Programming Serializable Classes” on page 5-8
Built-In Java Data Types

Cloudscape comes with several Java data types already built into the system and used in the system tables. These built-in Java data types come with class aliases and have public interfaces available to you. For more information, see “Types Used in System Tables” on page A-4 and Chapter 4, “Cloudscape System Tables”.

Assignability

You can store any object that is assignable to the named class or interface in the column. In general, if Java allows a reference to the named class to point to an object, you can store that object in the column. For example, you can store a subclass of the named class in the column.

Example

```sql
CREATE TABLE Cities
    (city SERIALIZE(JBMSTours.serializabletypes.City) )

-- The dynamic parameter ? is expected to be
-- a JBMSTours.City object when the INSERT is executed.
-- use setObject in Java to set the value for the question mark
INSERT INTO Cities VALUES(?)

-- Use a method call on a stored object:
SELECT city.isTropical()
FROM Cities
WHERE city.getName() = 'Seattle'
```

Implications of Serialization

For information about serialization and the java.io.Serializable API, see “Programming Serializable Classes” on page 5-8 in the Cloudscape Developer’s Guide.

When an object stored in the database is serialized into memory, the resulting in-memory object is a copy of the object in the database. Subsequent serializations of the now-in-memory object make separate copies of the object. Changes that are made to a copy of an object through method invocation are not automatically written back to the database. You can update an object stored in a column using the UPDATE statement, or delete the old row and insert a new row with a new object in its place, but doing so does not affect the value of any copy of the original object that is currently in memory.
Cloudscape stores the serialized object, not the Java class definition. If you change a class definition, the next time you start Cloudscape, Cloudscape uses the new class definition for method invocation.

Modifying classes that are used for column types must follow Java’s serialization rules for such modifications. Changes that do not follow these rules generate the same exceptions that come out of serialization. For further information about versioning of Java classes, see “Modifying Classes (Java Versioning)” on page 5-11 in the *Cloudscape Developer’s Guide*.

**Non-Serializable Java Data Types**

Cloudscape implicitly requires that Java data types be serializable. In most cases, a Java data type must be serializable:

- to be stored in a column
- to be returned to a remote client in a client/server environment
- to be used in any operation that requires sorting, such as aggregates, the DISTINCT operation, and self-referencing inserts and updates

Cloudscape enforces the serialization requirement only for values stored in tables. It does not enforce the serialization requirement for other values. For example, the following statement does not generate an error, even though the object is not serializable:

```
VALUES NEW java.lang.Object()
```

In an embedded environment, the calling application would be able to access the value returned by the statement, because it happens to be running in the same JVM as Cloudscape. However, in a client/server environment, the application would not be able to access the value returned by the statement.

It is technically possible to work with non-serializable objects, and sometimes useful. For example, you might want use Cloudscape’s virtual table interface to work with `java.awt.Images` and then display them locally in Cloudview. This is a very rare case, and, of course, you would not be able to store them, sort them, or access them remotely.

You can always execute methods on non-serializable objects that return built-in or serializable data types. See “Method Invocation” on page 1-231.

**Java Data Types and Equality Operations**

The operators = and <> are allowed on all Java data types.
NOTE: Correct results require that the Java data types override `equals(Object o)` and `hashCode()` from `java.lang.Object`. If a Java data type does not correctly implement these methods, the results of expressions using the `=` and `<>` operators with objects of that type are not guaranteed to be correct.

For more information, see “Notes on Implementing the equals and hashCode Methods” on page 5-43 in the Cloudscape Developer’s Guide.

NEW: The ability to use the operators `=` and `<>` on Java data types is new in Version 3.0.

To perform the equality or non equality operation, Cloudscape calls the either of the following methods off the object:

- `equals (Object o)` if the Java data type is not orderable.
- `compareTo (Object o)` if the Java data type is orderable

(A description of orderable Java data types follows).

### Orderable Java Data Types

Orderable Java data types are Java data types that fulfill some minimal requirements. If a Java data type is orderable, you can perform some SQL ordering and comparison operations on them and define indexes on them. (Those requirements are listed in “Programming Orderable Classes” on page 5-41 in the Cloudscape Developer’s Guide). You cannot perform these operations on columns that store non-orderable Java data types.

NEW: Orderable Java data types are new in Version 3.0.

(In the examples that follow, `customized_tour` stores a non-orderable Java data type, and `city` an orderable Java data type.)

These operations include the following items:

- Ordering and MAX/MIN aggregate operations
  Ordering operations include DISTINCT, GROUP BY, UNION, and ORDER BY.

  ```
  -- doesn’t work; the Java data type is not orderable
  SELECT customized_tour
  FROM CustomizedTours
  ORDER BY customized_tour
  ```
-- works; the Java data type is orderable
SELECT city FROM Cities
ORDER BY city

-- works
SELECT MAX(city) FROM Cities

Note that you can always order by a method call if it returns a built-in or orderable data type, however.

SELECT DISTINCT customized_tour.toString()
FROM CustomizedTours

• Comparisons <, >, <=, >=
  -- doesn't work; the Java data type is not orderable
  SELECT * FROM CustomizedTours WHERE customized_tour < ?
  -- works; the type is orderable
  SELECT city
  FROM Cities
  WHERE city < (SELECT city FROM cities WHERE city_id = 35)

• Creating indexes or primary, foreign key, or unique constraints
  A non-orderable Java data type cannot be the declared type for a column in an index, primary key, foreign key or unique constraint. For example, the following statement throws an exception:
  -- doesn't work; the type is not orderable
  CREATE TABLE Tours (tour SERIALIZE(tour) PRIMARY KEY)
  -- works
  CREATE TABLE Cities (city SERIALIZE(City) PRIMARY KEY)

Orderable Java Data Types and Indexes

NOTE: As stated in “Programming Orderable Classes” on page 5-41 in the Cloudscape Developer’s Guide, orderable Java data types require correct implementation of the compareTo, equals, and hashCode methods. If those methods are not implemented correctly, Cloudscape does not guarantee the correctness of indexes created on those types. Incorrect indexes lead to bizarre behavior and incorrect results. If you suspect a problem with the implementation of any of these methods, drop the index first. Re-create the index only when you are sure the methods are implemented correctly.
You cannot have indexes on columns that store orderable Java data types if those Java classes are loaded from the database. They must be loaded from the class path, instead.

**NOTE:** Cloudscape indexes have the restriction that the length of the key columns for an index must be less than half the page size of the index. Be sure to make the page size of an index large enough to accommodate this restriction.

```sql
CREATE BTREE INDEX c1 ON Cities(City)
PROPERTIES cloudscape.storage.pageSize=16384
```
SQL-J Expressions

Syntax for many statements and expressions includes the term Expression, or a term for a specific kind of expression such as Method Invocation or TableSubquery. Expressions are allowed in these specified places within statements. Some locations allow only a specific type of expression or one with a specific property. Table 1-7, “Table of Expressions,” on page 1-217, lists all the possible SQL-J expressions and indicates where they are allowed.

If not otherwise specified, an expression is permitted anywhere the word Expression appears in the syntax. This includes:

- SelectExpression
- UPDATE statement (SET portion)
- VALUES expression
- WHERE clause

Of course, many other statements include these elements as building blocks, and so allow expressions as part of these elements.

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General expressions</strong></td>
<td>All expressions that might result in a value of any type.</td>
</tr>
<tr>
<td>Column reference</td>
<td>A ColumnName that references the value of the column made visible to the expression containing the Column reference. You must qualify the ColumnName by the table name or correlation name if it is ambiguous. The qualifier of a ColumnName must be the correlation name, if a correlation name is given to a table that is in a FROM clause. The table name is no longer visible as a ColumnName qualifier once it has been aliased by a correlation name.</td>
</tr>
<tr>
<td>Literal</td>
<td>Most built-in data types typically have literals associated with them (as shown in “Data Types” on page 1-180).</td>
</tr>
</tbody>
</table>
### NULL

Allowed in CAST expressions or in INSERT VALUES lists and UPDATE SET clauses. Using it in a CAST expression gives it a specific data type.

NULL is an untyped literal representing the unknown value.

### Dynamic parameter

Allowed anywhere in an expression where the data type can be easily deduced. See “Dynamic Parameters” on page 1-226.

A dynamic parameter is a parameter to an SQL statement for which the value is not specified when the statement is created. Instead, the statement has a question mark (?) as a placeholder for each dynamic parameter. See “Dynamic Parameters” on page 1-226.

Dynamic parameters are permitted only in prepared statements. You must specify values for them before the prepared statement is executed. The values specified must match the types expected.

### CAST expression

Let's you specify the type of NULL or of a dynamic parameter or convert a value to another type. See “CAST” on page 1-134.

### scalar subquery

Subquery that returns a single row with a single column. See “ScalarSubquery” on page 1-101.

### table subquery

Allowed as a tableExpression in a FROM clause and with EXISTS, IN, and quantified comparisons.

Subquery that returns more than one column and more than one row. See “TableSubquery” on page 1-119.

### Method invocation

Invocation of a method associated with a Java class or instance of a class. See “Method Invocation” on page 1-231.

### Field access

Access of the public field of a Java class or instance of a class. See “Field Access” on page 1-238.

### Conditional expression

A conditional expression chooses an expression to evaluate based on a boolean test. See “Conditional (?:)” on page 1-145.

### User-Defined Aggregate

Performs a user-defined aggregation and can return any type. See “User-Defined Aggregate” on page 1-179.
### Boolean expressions
Expressions that result in boolean values. Most general expressions can result in boolean values.

Boolean expressions commonly used in a WHERE clause are made of operands operated on by SQL-J operators. See Table 1-8, “SQL-J Boolean Operators,” on page 1-223.

### Numeric expressions
Expressions that result in numeric values. Most of the general expressions can result in numeric values.

Numeric values have one of the following types:
- TINYINT, SMALLINT, INTEGER, LONGINT, REAL, DOUBLE PRECISION, DECIMAL.

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boolean expressions</strong></td>
<td>Expressions that result in boolean values. Most general expressions can result in boolean values. Boolean expressions commonly used in a WHERE clause are made of operands operated on by SQL-J operators. See Table 1-8, “SQL-J Boolean Operators,” on page 1-223.</td>
</tr>
<tr>
<td><strong>Numeric expressions</strong></td>
<td>Expressions that result in numeric values. Most of the general expressions can result in numeric values. Numeric values have one of the following types: TINYINT, SMALLINT, INTEGER, LONGINT, REAL, DOUBLE PRECISION, DECIMAL.</td>
</tr>
<tr>
<td>*<em>+, -, <em>, /, unary + and - expressions</em></em></td>
<td>Evaluate the expected math operation on the operands. If both operands are the same type, the result type is not promoted, so the division operator on integers results in an integer that is the truncation of the actual numeric result. When types are mixed, they are promoted as described in “Data Types” on page 1-180. Unary + is a noop (i.e., +4 is the same as 4). Unary - is the same as multiplying the value by -1, effectively changing its sign.</td>
</tr>
<tr>
<td><strong>AVG</strong></td>
<td>Returns the average of a set of numeric values. “AVG” on page 1-131</td>
</tr>
<tr>
<td><strong>SUM</strong></td>
<td>Returns the sum of a set of numeric values. “SUM” on page 1-174</td>
</tr>
<tr>
<td><strong>CHAR_LENGTH, CHARACTER_LENGTH</strong></td>
<td>Returns the number of characters in a character or bit string. See “CHAR_LENGTH, CHARACTER_LENGTH” on page 1-143.</td>
</tr>
<tr>
<td><strong>LOWER</strong></td>
<td>See “LOWER” on page 1-153.</td>
</tr>
<tr>
<td><strong>OCTET_LENGTH</strong></td>
<td>Returns the number of octets in a character string or bit string. See “OCTET_LENGTH” on page 1-159.</td>
</tr>
<tr>
<td><strong>BIT_LENGTH</strong></td>
<td>Returns the number of bits in a character string or bit string. See “BIT_LENGTH” on page 1-133.</td>
</tr>
</tbody>
</table>
### Table 1-7 Table of Expressions (continued)

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTRACT</td>
<td>Extracts DATE, TIME, or TIMESTAMP field as an integer value from a date/time data type. See “EXTRACT” on page 1-152.</td>
</tr>
<tr>
<td>COUNT</td>
<td>Returns the count of a set of values. See “COUNT” on page 1-146, “COUNT(*)” on page 1-147.</td>
</tr>
<tr>
<td>Character expressions</td>
<td>Expressions that result in a CHAR or VARCHAR value. Most general expressions can result in a CHAR or VARCHAR value.</td>
</tr>
<tr>
<td>A CHAR or VARCHAR value that uses wildcards.</td>
<td>The wildcards % and _ make a character string a pattern against which the LIKE operator can look for a match. See “LIKE” in Table 1-8 on page 1-224.</td>
</tr>
<tr>
<td>Concatenation expression</td>
<td>In a concatenation expression, the concatenation operator, “</td>
</tr>
<tr>
<td>USER functions</td>
<td>User functions return information about the current user as a String. See “CURRENT_USER” on page 1-151, “SESSION_USER” on page 1-169, and “USER” on page 1-178</td>
</tr>
<tr>
<td>Date/time expressions</td>
<td>A date/time expression results in a DATE, TIME, or TIMESTAMP value. Most of the general expressions can result in a date/time value.</td>
</tr>
<tr>
<td>CURRENT_DATE</td>
<td>Returns the current date. See “CURRENT_DATE” on page 1-148.</td>
</tr>
<tr>
<td>CURRENT_TIME</td>
<td>Returns the current time. See “CURRENT_TIME” on page 1-149.</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>Returns the current timestamp. See “CURRENT_TIMESTAMP” on page 1-150.</td>
</tr>
</tbody>
</table>
### Expression Precedence

Precedence of operations from highest to lowest is:

- (), ?, Literal (including sign), NULL, ColumnReference, ScalarSubquery, CAST, NEW, method invocation on columns
- CLASS, -> (field access), EXTRACT, CHAR_LENGTH, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, BIT_LENGTH, OCTET_LENGTH, and other built-ins
- method invocation (except on columns)
- unary + and -
- *, /, || (concatenation)
- binary + and -
- comparisons, Quantified comparisons, EXISTS, IN, IS NULL, LIKE, BETWEEN, INSTANCEOF, IS
- NOT
- AND
- OR
- ?: (Conditional)

You can explicitly specify precedence by placing expressions within parentheses. An expression within parentheses is evaluated before any operations outside the parentheses are applied to it.
Example

\((3+4)*9\)

\((age < 16 \text{ OR } age > 65) \text{ AND } employed = TRUE\)
**Boolean expression**

Boolean expressions are allowed in a number of places, most notably in WHERE clauses, but also in check constraints and VALUES expressions. Boolean expressions in check constraints have limitations not noted here; see “CONSTRAINT clause” on page 1-29 for more information. Boolean expressions in a WHERE clause have a highly liberal syntax; see “WHERE clause” on page 1-125, for example.

A boolean expression can include a boolean operator or operators. These are listed in Table 1-8 on page 1-223.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation and Example</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND, OR, NOT</td>
<td>Evaluate any operand(s) that are boolean expressions</td>
<td>{ Expression AND Expression</td>
</tr>
<tr>
<td>Comparisons</td>
<td>&lt;, =, &gt;, &lt;=, &gt;=, &lt;&gt; are applicable to all of the built-in types.</td>
<td>Expression {</td>
</tr>
<tr>
<td>IS NULL, IS NOT NULL</td>
<td>Test whether the result of an expression is null or not.</td>
<td>Expression IS [ NOT ] NULL</td>
</tr>
</tbody>
</table>

Table 1-8 SQL-J Boolean Operators

Cloudscape Version 3.0
### Table 1-8 SQL-J Boolean Operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation and Example</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIKE</td>
<td>Attempts to match a character expression to a character pattern, which is a character string that includes one or more wildcards.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>% matches any number (zero or more) of characters in the corresponding position in first character expression.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>_ matches one character in the corresponding position in the character expression.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any other character matches only that character in the corresponding position in the character expression.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>city.getName() LIKE ‘Sant_’</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CharacterExpression [ NOT ] LIKE CharacterExpression WithWildCard</td>
<td></td>
</tr>
<tr>
<td>BETWEEN</td>
<td>Tests whether the first operand is between the second and third operands. The second operand must be less than the third operand. Applicable only to types to which &lt;= and &gt;= can be applied.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE booking_date BETWEEN DATE’1998-02-26’ AND DATE’1998-03-01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Expression [ NOT ] BETWEEN Expression AND Expression</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>Operates on table subquery or list of values. Returns TRUE if the left expression’s value is in the result of the table subquery or in the list of values. Table subquery can return multiple rows but must return a single column.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE booking_date NOT IN (SELECT booking_date FROM HotelBookings WHERE rooms_available = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{ Expression [ NOT ] IN TableSubquery</td>
<td>Expression [ NOT ] IN ( Expression [, Expression ]*) }</td>
</tr>
<tr>
<td>EXISTS</td>
<td>Operates on a table subquery. Returns TRUE if the table subquery returns any rows, and FALSE if it returns no rows. Table subquery can return multiple columns (only if you use * to denote multiple columns) and rows.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE EXISTS SELECT * FROM Flights WHERE dest_airport = ‘SFO’ AND orig_airport = ‘GRU’)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EXISTS TableSubquery</td>
<td></td>
</tr>
</tbody>
</table>
Table 1-8 SQL-J Boolean Operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation and Example</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>Allows you to test whether a boolean expression returns TRUE, FALSE, or UNKNOWN. For example, for the following expression: c=3 it returns TRUE if c is 3, FALSE if c is any other value, and UNKNOWN if c is NULL. Useful in constraints. See “CONSTRAINT clause” on page 1-29.</td>
<td>BooleanExpression IS [NOT] { TRUE</td>
</tr>
<tr>
<td>Quantified comparison</td>
<td>A quantified comparison is a comparison operator (&lt;, =, &gt;, &lt;=, &gt;=, &lt;&gt;) with ALL or ANY or SOME applied. Operates on table subqueries, which can return multiple rows but must return a single column. If ALL is used, the comparison must be true for all values returned by the table subquery. If ANY or SOME is used, the comparison must be true for at least one value of the table subquery. ANY and SOME are equivalent. WHERE normal_rate &lt; ALL (SELECT budget/550 FROM Groups)</td>
<td>Expression ComparisonOperator { ALL</td>
</tr>
<tr>
<td>INSTANCEOF</td>
<td>Returns TRUE if the type of the Expression is an implementation or subtype of the JavaClassName type. If the Expression evaluates to NULL, the INSTANCEOF expression returns FALSE. See “INSTANCEOF Expression” on page 1-240.</td>
<td>Expression INSTANCEOF JavaClassName</td>
</tr>
</tbody>
</table>
Dynamic Parameters

You can prepare statements that are allowed to have parameters for which the value is not specified when the statement is prepared using PreparedStatement methods in the JDBC API. These parameters are called dynamic parameters and are represented by a ?.

The JDBC API documents refer to dynamic parameters as IN, INOUT, or OUT parameters. In SQL-J, they are always IN parameters.

You must specify values for them before executing the statement. The values specified must match the types expected.

Example

```java
PreparedStatement ps2 = conn.prepareStatement(
    "UPDATE HotelAvailability SET rooms_available = " +
    "(rooms_available - ?) WHERE hotel_id = ? " +
    "AND booking_date BETWEEN ? AND ?");
-- this sample code sets the values of dynamic parameters
-- to be the values of program variables
ps2.setInt(1, numberRooms);
ps2.setInt(2, theHotel.hotelId);
ps2.setDate(3, arrival);
ps2.setDate(4, departure);
updateCount = ps2.executeUpdate();
```

Where Dynamic Parameters Are Allowed

You can use dynamic parameters anywhere in an expression where their data type can be easily deduced.

1 Use as the first operand of BETWEEN is allowed if one of the second and third operands is not also a dynamic parameter. The type of the first operand is assumed to be the type of the non-dynamic parameter, or the union result of their types if both are not dynamic parameters.

```
WHERE ? BETWEEN DATE’1996-01-01’ AND ?
-- types assumed to be DATES
```

2 Use as the second or third operand of BETWEEN is allowed. Type is assumed to be the type of the left operand.

```
WHERE DATE’1996-01-01’ BETWEEN ? AND ?
-- types assumed to be DATES
```
3 Use as the left operand of an IN list is allowed if at least one item in the list is not itself a dynamic parameter. Type for the left operand is assumed to be the union result of the types of the non-dynamic parameters in the list.

```
WHERE ? NOT IN (?, ?, 'Santiago')
-- types assumed to be CHAR
```

4 Use in the values list in an IN predicate is allowed if the first operand is not a dynamic parameter or its type was determined in rule 3. Type of the dynamic parameters appearing in the values list is assumed to be the type of the left operand.

```
WHERE FloatColumn IN (?, ?, ?)
-- types assumed to be FLOAT
```

5 For the binary operators +, -, *, /, AND, OR, <, >, =, <=, and >=, use of a dynamic parameter as one operand but not both is permitted. Its type is taken from the other side.

```
WHERE ? < CURRENT_TIMESTAMP
-- type assumed to be a TIMESTAMP
```

6 Use in a CAST is always permitted. This gives the dynamic parameter a type.

```
CALL (CLASS java.lang.Integer).valueOf(CAST (? AS VARCHAR(10)))
```

7 Use on either or both sides of LIKE operator is permitted. When used on the left, the type of the dynamic parameter is set to the type of the right operand, but with the maximum allowed length for the type. When used on the right, the type is assumed to be of the same length and type as the left operand. (LIKE is permitted on CHAR and VARCHAR types; see “Concatenation” on page 1-144 and “LIKE” in Table 1-8 on page 1-224 for more information.)

```
WHERE ? LIKE ‘Santi%’
-- type assumed to be CHAR with a length of
-- java.lang.Integer.MAX_VALUE
```

8 A ? parameter is allowed by itself on only one side of the || operator. That is, “? || ?” is not allowed. The type of a ? parameter on one side of a || operator is determined by the type of the expression on the other side of the || operator. If the expression on the other side is a CHAR or VARCHAR, the type of the parameter is VARCHAR with the maximum
allowed length for the type. If the expression on the other side is a BIT or BIT VARYING, the type of the parameter is BIT VARYING with the maximum allowed length for the type.

```
SELECT BITcolumn || ?
FROM UserTable
-- Type assumed to be BIT of length specified for BITcolumn
```

9 In a conditional expression, which uses a ?, use of a dynamic parameter (which is also represented as a ?) is allowed. The type of a dynamic parameter as the first operand is assumed to be boolean. Only one of the second and third operands can be a dynamic parameter, and its type will be assumed to be the same as that of the other (that is, the third and second operand, respectively). (For more information about conditional expressions, see “Conditional (?:)” on page 1-145.)

```
SELECT c1 IS NULL ? ? : c1
-- allows you to specify a “default” value at execution time
-- dynamic parameter assumed to be the type of c1
-- you cannot have dynamic parameters on both sides
-- of the :
```

10 INSTANCEOF does not permit use of dynamic parameters.

11 Dynamic parameters are not permitted as receivers in a method invocation (see “Method Invocation” on page 1-231).

12 A dynamic parameter is allowed as an item in the values list or select list of an INSERT statement. The type of the dynamic parameter is assumed to be the type of the target column. A ? parameter is not allowed by itself in any select list, including the select list of a subquery, unless there is a corresponding column in a UNION (see no. 23, below) that is not dynamic.

```
INSERT INTO t VALUES (?)
-- dynamic parameter assumed to be the type
-- of the only column in table t
```

```
INSERT INTO t SELECT ?
FROM t2
-- not allowed
```

13 A ? parameter in a comparison with a subquery takes its type from the expression being selected by the subquery. For example:

```
SELECT *
FROM tab1
WHERE ? = (SELECT x FROM tab2)
```
SELECT *
FROM tab1
WHERE ? = ANY (SELECT x FROM tab2)
-- In both cases, the type of the dynamic parameter is
-- assumed to be the same as the type of tab2.x.

14 A dynamic parameter is allowed as the value in an UPDATE statement. The type of the dynamic parameter is assumed to be the type of the column in the target table.

UPDATE t2 SET c2 =? -- type is assumed to be type of c2

15 A dynamic parameter is allowed as the sole item in a WHERE clause. Its type is assumed to be boolean.

WHERE ?

16 A dynamic parameter is not allowed as the operand of the unary operators - or +.

17 A dynamic parameter is allowed as the operand of a NOT operator. Its type is assumed to be boolean.

WHERE NOT ?

18 A dynamic parameter as a method parameter is allowed if there is only one method with a matching name and number of parameters and that parameter type is not a Java primitive type.

VALUES (CLASS java.lang.Integer).toString(?)
-- not VALID. You need to use CAST (? as INT) instead of ?.

19 BIT_LENGTH and OCTET_LENGTH allow a dynamic parameter operand. It is assumed to be a maximum-length BIT VARYING type.

SELECT BIT_LENGTH(?)

20 CHAR_LENGTH and CHARACTER_LENGTH allow a dynamic parameter. The type is assumed to be a maximum length VARCHAR type.

SELECT CHAR_LENGTH(?)

21 EXTRACT does not allow any dynamic parameters.

22 Quantified comparisons.

? = SOME (SELECT 1 FROM t)
-- is valid. Dynamic parameter assumed to be INT type
1 = SOME (SELECT ? FROM t)
   -- is valid. Dynamic parameter assumed to be INT type.

23 A dynamic parameter is allowed to represent a column if it appears in a
UNION expression; Cloudscape can infer the data type from the
corresponding column in the UNION.

   SELECT ?
   FROM t
   UNION SELECT 1
   FROM t
   -- dynamic parameter assumed to be INT

   VALUES 1 UNION VALUES ?
   -- dynamic parameter assumed to be INT

24 A dynamic parameter is allowed as the left operand of an IS expression
and is assumed to be a boolean.

Once the type of a dynamic parameter is determined based on the expression it is
in, that expression is allowed anywhere it would normally be allowed if it did not
include a dynamic parameter. For example, above we said that a dynamic parameter
cannot be used as the operand of a unary -. It can, however, appear within an
expression that is the operand of a unary minus, such as:

   - (1+?)

The dynamic parameter is assumed to be an INT (because the binary operator +’s
other operand is of the type INT). Because we know its type, it is allowed as the
operand of a unary minus.

The use of a ? in a publication definition (for a Cloudscape synchronization system)
has a different form. For information, see the Cloudscape Synchronization Guide.

Java Expressions

This section discusses some of the Java extensions to SQL-J. Defining a column to
hold a Java data type is discussed in “CREATE TABLE statement” on page 1-48.
This section covers:

- “Method Invocation” on page 1-231
- “Field Access” on page 1-238
- “INSTANCEOF Expression” on page 1-240
- “NEW” on page 1-241
- “GETCURRENTCONNECTION()” on page 1-242
Method Invocation

Cloudscape allows you to invoke methods within SQL-J statements.

Method invocation invokes an object method or a class method. Object methods can be static or non-static (the first form of the syntax shown below), and are invoked on instances of Java classes. Class methods (the second form of the syntax shown below) are static, and are invoked on Java classes.

There is no requirement that all Java data types used by SQL-J be used in columns or be serializable. Only Java classes that are used in columns of tables created with CREATE TABLE must be serializable. You can invoke methods of non-serializable Java data types and pass the values they return to other Java methods.

Only public methods are accessible through SQL-J. Private, protected, and package methods are not accessible through SQL-J.

You can invoke methods that have a void return type only in the CALL statement (see “CALL statement” on page 1-25). To retrieve the value returned by a method, use a VALUES expression or a CALL statement using the ? = syntax (see “VALUES expression” on page 1-123) or if the method is a static method. Use a SELECT if the object is stored in the database (see “SelectExpression” on page 1-102).

Syntax

```
( Expression.JavaMethodName
  (Expression [, Expression]*)* )  |

{ (CLASS JavaClassName | ClassAlias).JavaMethodName |
  MethodAlias ]
  (Expression [, Expression]*) })
```

The expression to which the JavaMethodName is applied is called the method receiver.

ClassAlias is the name of a user-created or system-supplied alias for Java class. For more information, see “CREATE CLASS ALIAS statement” on page 1-39.

MethodAlias is the name of a user-created alias for a static Java method. For more information, see “CREATE METHOD ALIAS statement” on page 1-43.

NEW: Method invocation works with the new class alias construct. See “CREATE CLASS ALIAS statement” on page 1-39.
**Examples**

```sql
SELECT city.toString() FROM Cities

VALUES (CLASS COM.cloudscape.database.PropertyInfo).getDatabaseProperties()

-- PropertyInfo is an alias for
-- COM.cloudscape.database.PropertyInfo
VALUES PropertyInfo.getDatabaseProperties()

-- PropertyInfo is an alias for
-- COM.cloudscape.database.PropertyInfo
CALL PropertyInfo.setDatabaseProperty('', '')

VALUES NEW java.lang.Integer(3).shortValue()
```

**Type Conversion During Method Invocation**

You can invoke Java methods on SQL-J expressions. Every SQL-J expression has an SQL-J type; every SQL-J type has a corresponding Java class. This means that you can invoke methods of that corresponding class on SQL-J expressions and access the values they return. For example, you can use an SQL-J INTEGER column as a method receiver to invoke any method of `java.lang.Integer`. You can also use an SQL-J INTEGER type as a parameter to a method that takes a parameter of type `java.lang.Integer`.

For more information about the SQL-J to Java type conversion of method receivers and parameters during method invocation, see “SQL-J to Java Type Correspondence” on page 1-244.

The values returned by methods are converted to SQL-J types in most cases. For more information about the Java to SQL-J type conversion of return values, see “Java to SQL-J Type Correspondence” on page 1-245.

**Method Invocation and NULL**

A null return value from a method is converted to a SQL-J NULL value, and SQL-J NULL values are converted to Java null references.

**NEW:** In prior releases, using an SQL-J NULL value as a method receiver raised a runtime `NullPointerException`. This behavior goes away beginning in Version 3.0. However, if an SQL-J NULL value is used as a primitive method parameter, a `NullPointerException` is still thrown.
If the value of the instance specified in an instance method invocation is null, then the result of the invocation is NULL. For example:

```sql
SELECT city.getName() FROM Cities
```

If the `City` column is NULL, the `getName()` method invocation returns a NULL.

Primitive types in Java (boolean, char, byte, short, int, long, float, and double) cannot be null. So, when a method returning a primitive is called with a null receiver, or a null value is used to reference a non-static field, the field reference evaluates to null only if the value is being returned to the SQL domain. If the value is being returned to another Java construct, a `NullPointerException` is raised. For example, in the following SQL-J statement, where the `City` column is null, a null is returned:

```sql
SELECT city.showTemperature() FROM Cities
```

However, the situation changes when the return value of that method call is passed to another Java construct:

```java
SELECT NEW java.lang.Double(city.showTemperature()) FROM Cities
```

In this situation, if the `City` column contains any nulls, a `NullPointerException` is raised because the null value is passed to the Java constructor, which does not accept nulls.

Note that when a method invocation evaluates to null because its receiver is null, the method is not actually called, and its arguments are not evaluated. This only matters for methods or arguments that have side effects. (A side effect is anything the method does other than return a value using calculations based on its parameters. For example, changing the value of a field (static or non-static), doing I/O, and doing an update in the database are all side effects.)

### Static Method Access and Null Receivers

It is possible to call a `static` method, however, off of a null receiver. The method is actually called. For example:

```java
-- method is called even though receiver is null
CALL CAST (NULL AS HotelStay).makeHistoryDatabase()

-- method is called even though receiver is null
CALL (SELECT customized_tour.getStay1() FROM CustomizedTours where group_id = 0).makeHistoryDatabase()
```
-- the city column contains some nulls.
-- the non-static method call (getName()) returns NULL
-- when the column is null. HOWEVER, the static
-- method (findCity) gets called, returns a value,
-- and performs any side effects even if the
-- the column is null
SELECT country, countries.country_ISO_code, city.getName(),
city.findCity(getCurrentConnection(), 35)
FROM countries LEFT OUTER JOIN cities ON
countries.country_ISO_code = cities.country_ISO_code
WHERE countries.country_ISO_code LIKE 'V%'

Method Invocation and Dynamic Parameters (?)

For a statement to compile correctly, Cloudscape must be able to infer the data type
of a dynamic parameter (?).

Because of signature overloading, to use a dynamic parameter as a parameter to a
Java method, you usually must cast the data type of the ?.

For example:

CALL (CLASS java.lang.Integer).valueOf(CAST (? AS CHAR(10)))

There are a few exceptions. If there is only one method in the class with that name
and number of parameters, you do not need to cast.

If there is more than one method with the same name and number of parameters,
you do not need to cast if the other parameters that are not dynamic parameters (?)
serve to distinguish the method.

In either case, the dynamic parameter must not be a Java primitive. You must always
cast a dynamic parameter of a Java primitive data type to its corresponding SQL-J
type when using it as the parameter to a Java method.

It is a good idea always to cast the data type of a ? used as a parameter to a Java
method.

Method Invocation and Exceptions

A runtime exception that is thrown from a method is caught and translated to the
following statement exception:

The exception "(0)" was thrown from a user expression.

When the return value for a method is passed to another method, exceptions from
the first method are not caught and translated.
Only exceptions that are allowed to “escape” from user-written code into the rest of the query are caught and translated. Consider the following example:

```java
SELECT c1.catchesTheException(c2.throwsAnException())
FROM t
```

The exception thrown by the `throwsAnException` method is caught by the `catchesTheException` method. Since `catchesTheException` does not throw an exception itself, the query does not get the statement exception described above.

Methods invoked within SQL-J statements should not catch `SQLExceptions` of transaction severity or higher. If the `SQLException` is of transaction severity (for example, a deadlock), you will get unpredictable results. Instead, `SQLExceptions` of transaction severity or higher should be passed to the calling application. For more information, see “Database-Side JDBC Methods and SQLExceptions” on page 5-6 in the *Cloudscape Developer’s Guide*.

### Method Resolution

For information on method resolution, see “Method Resolution and Type Correspondence” on page 1-246.

### Method Invocation and INTEGER Literals

You cannot directly call a method off an INTEGER literal, since following an INTEGER literal with a period turns it into a DOUBLE PRECISION literal. Instead, you can put it in parentheses or turn it into a DOUBLE PRECISION literal with an additional period; for example:

```java
VALUES 1..toString()
VALUES (1).toString()
```

### Class Alias and Column Name Ambiguity

It is possible that a database has a class alias and a column of the same name and that a method call or field reference could be ambiguous. Where such duplication exists, Cloudscape first attempts to resolve the name as a column name. For example, in the following method call, `City` could be a class alias or a column:

```java
City.getName()
```

If `City` is a class alias, the method must be a static method. If `City` is a column name, then the method could be any public method. The same holds true for field references.
Usually, the context determines the resolution. If the statement is a SELECT, Cloudscape attempts to resolve City as a column name. It can do this if the table that contains the column City is in the FROM list. For example:

```
SELECT City.getName() FROM Cities
```

In this case, the method getName() can be either an object or a static class method. However, keep in mind that you don’t typically execute static methods for every row in a table, so it makes more sense if the method is a non-static method.

If the statement is not a SELECT, Cloudscape would not be able to resolve City as a column name, and it attempts to resolve it as a class alias:

```
VALUES City.getName()
```

In this case, the method getName() would have to be a static method (which, in the sample database, it is not).

In the unlikely situation that you alter a table to add or drop a column name that is the same as a class alias, existing statements are invalidated and may be recompiled with a different resolution, as described below.

Imagine a table called TropicalCities:

```
CREATE TABLE TropicalCities (ID int, Name VARCHAR(85))
```

Your application also has a class called JBMSTours.serializabletypes.City, for which you have created the alias City. This class has a static method called findCity. You would probably execute the method with the VALUES statement (and not a SELECT statement), something like this:

```
VALUES City.findCity(getCurrentConnection(), 3)
```

However, you could conceivably execute it within a SELECT statement, in which case the method would be executed once for every row returned (which probably wouldn’t make a lot of sense unless you passed in one of the columns somehow has an argument):

```
SELECT name, City.findCity(getCurrentConnection(), ID) FROM TropicalCities
```

Suppose that after compiling this statement, you alter the TropicalCities table and add a column called City:

```
ALTER TABLE TropicalCities ADD COLUMN city SERIALIZE(City)
```

Such a change would invalidate the existing statement and cause it to be recompiled upon the next execution. When Cloudscape recompiles the statement, it would notice the ambiguity in the name City in the select list and would interpret it as a
column name instead of an alias name. The statement may then return very different results.
Field Access

You can access the field of an object or class. Object fields can be static or non-static, and are accessed from instances of Java classes. Class fields are static, and are accessed from Java classes.

Only public fields are accessible through SQL-J. Private, protected, and package fields are not accessible through SQL-J.

The fields that can be accessed are those that are declared as public in the Java class of the expression’s declared data type. The “declared” data type of an expression is its type as determined by the declared types of columns, the declared return types of methods, and the expected result types of the various SQL-J expressions based on the operand(s) involved in the expressions.

Syntax

```sql
{  
    Expression->JavaFieldName  |  
    [(CLASS JavaClassName) | ClassAlias ]->JavaFieldName  
}
```

NEW: Field access works with the new class alias construct. See “CREATE CLASS ALIAS statement” on page 1-39.

Example

VALUES (CLASS java.lang.Integer)->MAX_VALUE

-- Accessing a static field
-- Tour is a class alias
VALUES Tour->FIRSTCLASSTOURLEVEL

-- access a non-static field
SELECT customized_tour->begin FROM CustomizedTours

Interaction with Java Data Types

You can perform field accesses on any SQL-J expression that evaluates Java object or class. Every SQL-J expression has an SQL-J type; every SQL-J type has a corresponding Java class. This means that you can perform field accesses on the corresponding class of SQL-J expressions, returning the result of the field access on that class. The field can be in the corresponding class or in any class within the superclass hierarchy.
Field Access and Type Conversions

The values returned by field accesses are converted to SQL-J types in most cases. For more information about the Java to SQL-J type conversion of return values, see “Java to SQL-J Type Correspondence” on page 1-245.

Field Access and Null Values

**NEW:** In prior releases, if an expression referenced a Java field that was null, the JVM threw a `NullPointerException`. Beginning in Version 3.0, a reference to a null Java field returns an SQL-J NULL.

Primitive types in java (boolean, char, byte, short, int, long, float, and double) cannot be null. So, when a null value is used to reference a non-static field, the field reference evaluates to null only if the value is being returned to the SQL domain. If the value is being returned to another Java construct, a `NullPointerException` is raised. For example, in the following SQL-J statement, where the `city` column is null, a null is returned:

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

However, the situation changes when the return value of that method call is passed to another Java construct:

```
SELECT NEW java.lang.String(city->language)
FROM Cities
```

In this situation, if the `City` column contains any nulls, a `NullPointerException` is raised because the null value is passed to the Java constructor, which does not accept nulls.

Static Field Access and Null Values

It is possible to access a static field, however, off of a null receiver. In the following example, the value of the static field is returned, not an SQL NULL:

```
VALUES CAST (NULL AS Tour)->ECONOMYTOURLEVEL
```

Class Alias and Column Name Ambiguity

See the discussion under “Method Invocation” on page 1-231.
INSTANCEOF Expression

An INSTANCEOF expression evaluates to a BOOLEAN. It returns TRUE if the type of the expression is an implementation or subtype of the JavaClassName type. If the expression evaluates to NULL, the INSTANCEOF expression returns FALSE.

Syntax

Expression INSTANCEOF [ JavaClassName | ClassAlias ]

NEW: The INSTANCEOF operator works with the new class alias construct. See “CREATE CLASS ALIAS statement” on page 1-39.

In the situation in which a class alias and a Java class name shared the same name, Cloudscape finds the Java class name first.

Example

SELECT * FROM People
WHERE person INSTANCEOF JBMSTours.serializabletypes.Adult

-- Adult is a class alias
SELECT * FROM People
WHERE person INSTANCEOF Adult

-- Verifying the type of a class’s field
SELECT customized_tour->begin INSTANCEOF java.sql.Date
FROM CustomizedTours
NEW

Creates a new instance of the specified class.

**Syntax**

```
NEW [ JavaClassName | ClassAlias ]
   ( Expression [ , Expression ]* )
```

You can place a NEW expression anywhere an expression is permitted. It invokes the constructor that matches the types of the expressions in the parameter list.

**NEW:** The NEW operator works with the new class alias construct. See “CREATE CLASS ALIAS statement” on page 1-39.

In the situation in which a class alias and a Java class name shared the same name, Cloudscape finds the Java class name first.

**Example**

```
INSERT INTO People VALUES (50, NEW JBMSTours.serializabletypes.Person(
   'George', 'Washington'))

-- Person is a ClassAlias
INSERT INTO People VALUES (50, NEW Person(
   'Martha', 'Washington'))
```

For information on method resolution, see “Method Resolution and Type Correspondence” on page 1-246.
**GETCURRENTCONNECTION()**

This built-in function returns the current connection (*java.sql.Connection*). It is used to pass the current connection to a method invoked within Cloudscape that makes JDBC calls back to the database. This simplifies the coding for user methods and allows you to invoke methods with the same signatures and code in both the client application and Cloudscape. For more information, see “Requirements for Database-Side JDBC Methods Using Nested Connections” on page 5-3 in the *Cloudscape Developer’s Guide*.

**Syntax**

```plaintext
GETCURRENTCONNECTION()
```

**Example**

```sql
SELECT city.getDistanceFrom(getCurrentConnection(), 35)
FROM Cities
```
SQL-J and Java Type Correspondence

This section covers the following topics:

- “Column Values and Type Correspondence” on page 1-243
- “SQL-J to Java Type Correspondence” on page 1-244
- “Java to SQL-J Type Correspondence” on page 1-245
- “Method Resolution and Type Correspondence” on page 1-246

Each built-in type in SQL-J has a Java class associated with it. For example, the corresponding type for an SQL-J INTEGER is `java.lang.Integer`. Consider the following SQL-J statement:

```
VALUES (1 INSTANCEOF java.lang.Integer)
```

It returns the value TRUE, since the SQL-J type INTEGER maps to the Java class `java.lang.Integer`.

Column Values and Type Correspondence

You can insert a value of a built-in type into a column of its corresponding Java class, and vice versa. Table 1-9, “Java Classes Associated with SQL-J Built-In Types”, shows the correspondence between types:

<table>
<thead>
<tr>
<th>Built-In SQL-J Type</th>
<th>Corresponding Java Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td><code>java.lang.Integer</code></td>
</tr>
<tr>
<td>SMALLINT</td>
<td><code>java.lang.Short</code></td>
</tr>
<tr>
<td>LONGINT</td>
<td><code>java.lang.Long</code></td>
</tr>
<tr>
<td>TINYINT</td>
<td><code>java.lang.Byte</code></td>
</tr>
<tr>
<td>REAL</td>
<td><code>java.lang.Float</code></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td><code>java.lang.Double</code></td>
</tr>
<tr>
<td>FLOAT</td>
<td><code>java.lang.Float OR java.lang.Double</code>, depending on the precision you specify</td>
</tr>
<tr>
<td>DECIMAL</td>
<td><code>java.math.BigDecimal</code></td>
</tr>
<tr>
<td>NUMERIC</td>
<td><code>java.math.BigDecimal</code></td>
</tr>
<tr>
<td>CHAR</td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td>VARCHAR</td>
<td><code>java.lang.String</code></td>
</tr>
</tbody>
</table>

Cloudscape Version 3.0
SQL-J Language Reference

**Table 1-9** Java Classes Associated with SQL-J Built-In Types (continued)

<table>
<thead>
<tr>
<th>Built-In SQL-J Type</th>
<th>Corresponding Java Class</th>
<th>Second-Chance Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONG VARCHAR</td>
<td><code>java.lang.String</code></td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td><code>byte[]</code></td>
<td></td>
</tr>
<tr>
<td>BIT VARYING</td>
<td><code>byte[]</code></td>
<td></td>
</tr>
<tr>
<td>LONG VARBINARY</td>
<td><code>byte[]</code></td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td><code>java.lang.Boolean</code></td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td><code>java.sql.Date</code></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td><code>java.sql.Time</code></td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td><code>java.sql.Timestamp</code></td>
<td></td>
</tr>
</tbody>
</table>

**SQL-J to Java Type Correspondence**

Table 1-10, “Conversion of SQL-J Types to Java Classes During Method Invocation”, shows the correspondence between SQL-J types and Java classes. SQL-J expressions that become method receivers, as well as SQL-J expressions used as parameter values passed to the method, are mapped to Java types according to this table during method invocation.

The third column shows what primitive is used during second-chance conversion by Cloudscape if on the first pass it cannot find a method with the matching signature. This conversion is all-or-nothing; see “Method Resolution and Type Correspondence” on page 1-246.

**Table 1-10** Conversion of SQL-J Types to Java Classes During Method Invocation

<table>
<thead>
<tr>
<th>SQL-J Type</th>
<th>Corresponding Java Class</th>
<th>Second-Chance Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td><code>java.lang.Integer</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td>SMALLINT</td>
<td><code>java.lang.Short</code></td>
<td><code>short</code></td>
</tr>
<tr>
<td>LONGINT</td>
<td><code>java.lang.Long</code></td>
<td><code>long</code></td>
</tr>
<tr>
<td>TINYINT</td>
<td><code>java.lang.Byte</code></td>
<td><code>byte</code></td>
</tr>
<tr>
<td>REAL</td>
<td><code>java.lang.Float</code></td>
<td><code>float</code></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td><code>java.lang.Double</code></td>
<td><code>double</code></td>
</tr>
<tr>
<td>FLOAT</td>
<td><code>java.lang.Float OR java.lang.Double</code>, depending on the precision you specify</td>
<td><code>float/double</code></td>
</tr>
<tr>
<td>DECIMAL</td>
<td><code>java.math.BigDecimal</code></td>
<td><code>double</code></td>
</tr>
<tr>
<td>NUMERIC</td>
<td><code>java.math.BigDecimal</code></td>
<td><code>double</code></td>
</tr>
</tbody>
</table>
Java to SQL-J Type Correspondence

In SQL-J statements, Cloudscape converts the return values of invoked Java methods and accessed fields of Java objects or classes to SQL-J values, except when the value is used as a receiver or parameter to another method. In that case, the value remains a Java value. This means, for example, that if a method returns a Java int type, that int can be passed directly to a Java method that takes an int; it is not first converted to a SQL-J INTEGER and then back to a Java java.lang.Integer.

Cloudscape converts return values to SQL-J built-in types according to Table 1-11, “Conversion of Java Types to SQL-J Types”.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Corresponding SQL-J Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Integer</td>
<td>INTEGER</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>LONGINT</td>
</tr>
<tr>
<td>long</td>
<td>LONGINT</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>TINYINT</td>
</tr>
<tr>
<td>byte</td>
<td>TINYINT</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>REAL</td>
</tr>
</tbody>
</table>

Table 1-10 Conversion of SQL-J Types to Java Classes

<table>
<thead>
<tr>
<th>SQL-J Type</th>
<th>Corresponding Java Class</th>
<th>Second-Chance Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>java.lang.String</td>
<td></td>
</tr>
<tr>
<td>VARCHAR</td>
<td>java.lang.String</td>
<td></td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>java.lang.String</td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>byte[]</td>
<td></td>
</tr>
<tr>
<td>BIT VARYING</td>
<td>byte[]</td>
<td></td>
</tr>
<tr>
<td>LONG VARBINARY</td>
<td>byte[]</td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>java.lang.Boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td></td>
</tr>
<tr>
<td>Java data type</td>
<td>a.b.c.</td>
<td>a.b.c.</td>
</tr>
</tbody>
</table>
No Java types map to FLOAT, CHAR, BIT, LONG VARCHAR, or LONG VARCHAR by default. The Java primitive types char and byte do not map to any SQL-J types.

### Method Resolution and Type Correspondence

General *Method Invocation*, the NEW command constructor method invocation, and the CALL statement method invocation all require method resolution.

The rules of Java determine which method is invoked based on the method signature. The signature of a method is its name, the class of the receiver, and the number and types of the parameters. Cloudscape searches the receiving class for a method with a matching signature. Cloudscape can find methods in superclasses of the receiving class. Cloudscape uses the method with the signature that has the best match according to the rules of Java. The return type of the method is determined by the best matching method.

If on the first attempt Cloudscape does not find a match, Cloudscape converts all SQL-J numeric types and BOOLEANs to their corresponding Java primitive types. (This conversion is all-or-nothing.) The conversion from SQL-J to Java primitive types is done according to Table 1-12, “Second-Chance Conversion of SQL-J Types to Java Primitive Types,” on page 1-247.

Thus:

```java
NEW java.lang.Integer(1)
```
finds the correct constructor. On the first attempt, Cloudscape looks for a method with a `java.lang.Integer` parameter (because that is the corresponding type of the SQL-J INTEGER type) and fails. On the second attempt, Cloudscape looks for a method with a primitive `int` parameter and succeeds.

Cloudscape uses Java's rules for parameter broadening to determine whether a method matches an invocation. For example, consider the following method invocation:

```java
myColumn.myLongMethod((1).intValue())
```

The parameter is a `java.lang.Integer` (which is converted to an `int` on the second try) so the invocation can match a method that takes a long parameter.

When a primitive parameter is dynamic, you may need to CAST the parameter so that Cloudscape can determine its data type. For example:

```java
-- will get an error
CREATE STATEMENT j4 AS VALUES (CLASS java.lang.Math).abs(?)
ERROR 42X50: No method was found with the signature
java.lang.Math.abs(UNTYPED).
```

Since you cannot work with primitive Java dynamic types within SQL-J, how do you do that? CAST it to the corresponding SQL-J data type. Suppose that in the above example, you want to use the signature of Math.abs that takes a primitive float value. That primitive data type corresponds to the SQL-J REAL data type, so this works:

```java
CREATE STATEMENT j4 as
VALUES (CLASS java.lang.Math).abs(CAST (? AS REAL))
```

<table>
<thead>
<tr>
<th>SQL-J Types</th>
<th>Corresponding Java Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINYINT</td>
<td>byte</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>boolean</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>LONGINT</td>
<td>long</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>double</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>double</td>
</tr>
</tbody>
</table>
Complete List of Reserved Words

This section lists all the Cloudscape reserved words, including those in the SQL-92 standard. * indicates new reserved words in Version 3.0.

<table>
<thead>
<tr>
<th>Reserved Word</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSOLUTE</td>
<td>CALL</td>
</tr>
<tr>
<td>ACTION</td>
<td>CASCADE</td>
</tr>
<tr>
<td>ADD</td>
<td>CASCADED</td>
</tr>
<tr>
<td>ALL</td>
<td>CASE</td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>CAST</td>
</tr>
<tr>
<td>ALTER</td>
<td>CATALOG</td>
</tr>
<tr>
<td>AND</td>
<td>CHAR</td>
</tr>
<tr>
<td>ANY</td>
<td>CHAR_LENGTH</td>
</tr>
<tr>
<td>ARE</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>AS</td>
<td>CHARACTER_LENGTH</td>
</tr>
<tr>
<td>ASC</td>
<td>CHECK</td>
</tr>
<tr>
<td>ASSERTION</td>
<td>CLOSE</td>
</tr>
<tr>
<td>AT</td>
<td>COALESCE</td>
</tr>
<tr>
<td>AUTHORIZATION</td>
<td>COLLABATE</td>
</tr>
<tr>
<td>AVG</td>
<td>COLLATION</td>
</tr>
<tr>
<td>BEGIN</td>
<td>COLUMN</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>COMMIT</td>
</tr>
<tr>
<td>BIT</td>
<td>CONNECT</td>
</tr>
<tr>
<td>BIT_LENGTH</td>
<td>CONNECTION</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>CONSTRAINT</td>
</tr>
<tr>
<td>BOTH</td>
<td>CONSTRAINTS</td>
</tr>
<tr>
<td>BY</td>
<td>CONTINUE</td>
</tr>
</tbody>
</table>
CONVERT
CORRESPONDING
COUNT
CREATE
CROSS
CURRENT
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
CURSOR
DATE
DAY
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DEFERRABLE
DEFERRED
DELETE
DESC
DESCRIBE
DIAGNOSTICS
DISCONNECT
DISTINCT
DOMAIN
DOUBLE
DROP
ELSE
END
ENDEXEC
EXTERNAL
EXTRACT
FALSE
FETCH
FIRST
FLOAT
FOR
FOREIGN
FOUND
FROM
FULL
GET
GETCURRENTCONNECTION
GLOBAL
GO
GOTO
GRANT
GROUP
HAVING
HOUR
IDENTITY
IMMEDIATE
IN
INSENSITIVE
INSERT
INT
INTEGER
INTERSECT
INTO
IS
ISOLATION
JOIN
KEY
LANGUAGE
LAST
LEADING
LEFT
LEVEL
LIKE
LOCAL
LONGINT
LOWER
LTRIM
MATCH
MAX
MIN
MINUTE
MODULE
MONTH
NATIONAL
NATURAL
NCHAR
NEXT
NULL
NULLIF
NUMERIC
OCTET_LENGTH
OF
ON
<table>
<thead>
<tr>
<th>Reserved Word</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONLY</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>OPEN</td>
<td>SUBSTR</td>
</tr>
<tr>
<td>OPTION</td>
<td>SUBSTRING</td>
</tr>
<tr>
<td>OR</td>
<td>SUM</td>
</tr>
<tr>
<td>ORDER</td>
<td>SYSTEM_USER</td>
</tr>
<tr>
<td>OUTER</td>
<td>TABLE</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>TEMPORARY</td>
</tr>
<tr>
<td>OVERLAPS</td>
<td>THEN</td>
</tr>
<tr>
<td>PAD</td>
<td>TIME</td>
</tr>
<tr>
<td>PARTIAL</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>POSITION</td>
<td>TIMEZONE_HOUR</td>
</tr>
<tr>
<td>PREPARE</td>
<td>TIMEZONE_MINUTE</td>
</tr>
<tr>
<td>PRESERVE</td>
<td>TINYINT</td>
</tr>
<tr>
<td>PRIMARY</td>
<td>TO</td>
</tr>
<tr>
<td>PRIOR</td>
<td>TRAILING</td>
</tr>
<tr>
<td>PRIVILEGES</td>
<td>TRANSACTION</td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>TRANSLATION</td>
</tr>
<tr>
<td>READ</td>
<td>TRIM</td>
</tr>
<tr>
<td>REAL</td>
<td>TRUE</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>UNION</td>
</tr>
<tr>
<td>RELATIVE</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>RESTRICT</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>REVOKE</td>
<td>UPDATE</td>
</tr>
<tr>
<td>RIGHT</td>
<td>UPPER</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>USAGE</td>
</tr>
<tr>
<td>ROWS</td>
<td>USER</td>
</tr>
<tr>
<td>RTRIM</td>
<td>USING</td>
</tr>
<tr>
<td>RUNTIMESTATISTICS</td>
<td>VALUES</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>SCROLL</td>
<td>VARYING</td>
</tr>
<tr>
<td>SECOND</td>
<td>VIEW</td>
</tr>
<tr>
<td>SECTION</td>
<td>WHEN</td>
</tr>
<tr>
<td>SELECT</td>
<td>WHenever</td>
</tr>
<tr>
<td>SESSION</td>
<td>WHERE</td>
</tr>
<tr>
<td>SESSION_USER</td>
<td>WITH</td>
</tr>
<tr>
<td>SET</td>
<td>WORK</td>
</tr>
<tr>
<td>SIZE</td>
<td>WRITE</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>YEAR</td>
</tr>
<tr>
<td>SOME</td>
<td>ZONE</td>
</tr>
<tr>
<td>SPACE</td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td></td>
</tr>
<tr>
<td>SQLCODE</td>
<td></td>
</tr>
<tr>
<td>SQLERROR</td>
<td></td>
</tr>
</tbody>
</table>
Nonreserved Keywords

This section lists all the Cloudscape keywords, including those in the SQL-92 standard, except for the words that are reserved, which are listed above.

ADA
AFTER*
AGGREGATE*
ALIAS
BEFORE*
BINARY
C
CLASS
COBOL
COPY* (formerly reserved)
COMMITTED
D
DATA
DATABASE
DISABLED
EACH*
ENABLED
EXCLUSIVE
FORTRAN
INDEX
INSTANCEOF
INTERVAL
INVALID*
LENGTH
JAR
LOCK
LONG
METHOD
MODE
MORE
MUMPS
NAME
NEW
NOCOMPILE*
NULLABLE
NUMBER

OLD*
OJ
OFF
PARAMETER*
PASCAL
PLI
PRECISION
PROPERTY
RECOMPILE*
REPEATABLE
REREFERENCING*
ROW*
SCALE
SERIALIZE
SERIALIZABLE
SHARE
STATISTICS
STATEMENT
TIMING
T
TS
TARGET
TRIGGER*
TRIGGERS*
TYPE
UNCOMMITTED
VALUE
VARBINARY
FILE
FILTER
PUBLICATION
RECOMMILE
REFRESH
WAIT
Cloudscape 3.0 Support for SQL-92 Features

Table 3-1 shows the SQL-92 features that Cloudscape 3.0 supports. There are four levels of SQL-92 support:

- SQL92E
  Entry
- SQL92T
  Transitional, a level defined by NIST in a publication called FIPS 127-2
- SQL92I
  Intermediate
- SQL92F
  Full

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic types</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>NUMERIC(p,s)</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>REAL</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>FLOAT(p)</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td><strong>Basic math operations</strong></td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>+, *, -, /, unary +, unary -</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 3-1 Support for SQL-92 Features (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Basic comparisons</em></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td><code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>, <code>&lt;&gt;</code>, <code>=</code></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td><em>Basic predicates</em></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>BETWEEN, LIKE, NULL</td>
<td></td>
<td>yes (except ESCAPE clause on LIKE)</td>
</tr>
<tr>
<td><em>Quantified predicates</em></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>IN, ALL/SOME, EXISTS</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td><em>schema definition</em></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>tables</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>views</td>
<td></td>
<td>yes (not updatable)</td>
</tr>
<tr>
<td>privileges</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td><em>column attributes</em></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>default values</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>nullability</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td><em>constraints (non-deferrable)</em></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>NOT NULL</td>
<td></td>
<td>yes (not stored in SYSCONSTRAINTS)</td>
</tr>
<tr>
<td>UNIQUE/PRIMARY KEY</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>CHECK</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>View WITH CHECK OPTION</td>
<td></td>
<td>no, since views are not updatable</td>
</tr>
<tr>
<td>Delimited identifiers</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Correlated subqueries</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td><em>Cursors</em></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>DECLARE, OPEN, FETCH, CLOSE</td>
<td></td>
<td>done through JDBC</td>
</tr>
<tr>
<td>UPDATE, DELETE CURRENT</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Insert, Update, Delete statements</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Joins</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Where qualifications</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Group by</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Having</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Aggregate functions</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Order by</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
</tbody>
</table>
### Table 3-1 Support for SQL-92 Features (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select expressions</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Select *</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>SQL92E</td>
<td>no, deprecated in SQL-92</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Dynamic SQL 1</strong></td>
<td>SQL92T</td>
<td>done through JDBC</td>
</tr>
<tr>
<td>ALLOCATE/DEALLOCATE/GET/SET DESCRIPTOR</td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>PREPARE/EXECUTE/EXECUTE IMMEDIATE</td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>DECLARE, OPEN, FETCH, CLOSE, UPDATE, DELETE dynamic cursor</td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>DESCRIBE output</td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td><strong>Basic information schema</strong></td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>TABLES</td>
<td>SYS.SYSTABLES, SYS.SYSVIEWS, SYS.SYSCOLUMNS</td>
<td></td>
</tr>
<tr>
<td>VIEWS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLUMNS</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Basic schema manipulation</strong></td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>CREATE/DROP TABLE</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CREATE/DROP VIEW</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>GRANT/REVOKE</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>ALTER TABLE ADD COLUMN</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>ALTER TABLE DROP COLUMN</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td><strong>Joined table</strong></td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>INNER JOIN</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>natural join</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>LEFT, RIGHT OUTER JOIN</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>join condition</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>named columns join</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>
### Cloudscape 3.0 Support for SQL-92 Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATETIME data types</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simple DATE, TIME, TIMESTAMP, INTERVAL</td>
<td>SQL92T</td>
<td>not INTERVAL</td>
</tr>
<tr>
<td>datetime literals</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>datetime math</td>
<td></td>
<td>can use Java methods</td>
</tr>
<tr>
<td>datetime comparisons</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>predicates: OVERLAPS</td>
<td></td>
<td>no, can do with Java methods</td>
</tr>
<tr>
<td><strong>VARCHAR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LENGTH</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>SUBSTRING</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>concatenation (</td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>TRIM function</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>UNION in views</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Implicit numeric casting</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Implicit character casting</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td><strong>Transaction isolation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ WRITE/READ ONLY</td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>Synchronization targets can be read-only (set when created). User connections can be read-only. You can also use databases on read-only media.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RU, RC, RR, SER</td>
<td></td>
<td>RC, SER, and RR (maps to SER)</td>
</tr>
<tr>
<td>Get diagnostics</td>
<td>SQL92T</td>
<td>use JDBC SQLExceptions</td>
</tr>
<tr>
<td>Grouped operations</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Qualified * in select list</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Lowercase identifiers</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>nullable PRIMARY KEYs</td>
<td>SQL92T</td>
<td>no</td>
</tr>
<tr>
<td><strong>Multiple schemas per user</strong></td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>SCHEMATA view</td>
<td></td>
<td>SYS.SYSSCHEMAS</td>
</tr>
</tbody>
</table>

**Table 3-1** Support for SQL-92 Features (continued)
### Cloudscape 3.0 Support for SQL-92 Features

**Table 3-1** Support for SQL-92 Features (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple module support</td>
<td>SQL92T</td>
<td>no (not required and not part of JDBC)</td>
</tr>
<tr>
<td>Referential delete actions</td>
<td>SQL92T</td>
<td>only RESTRICT, or use triggers</td>
</tr>
<tr>
<td>CAST functions</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>INSERT expressions</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit defaults</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Privilege tables</strong></td>
<td>SQL92T</td>
<td>no</td>
</tr>
<tr>
<td>TABLE_PRIVILEGES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLUMNS_PRIVILEGES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USAGE_PRIVILEGES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keyword relaxations</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Domain definition</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>CASE expression</td>
<td>SQL92I</td>
<td>use conditional expression</td>
</tr>
<tr>
<td>Compound character literals</td>
<td>SQL92I</td>
<td>use concatenation</td>
</tr>
<tr>
<td>LIKE enhancements</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>UNIQUE predicate</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td><strong>Table operations</strong></td>
<td>SQL92I</td>
<td></td>
</tr>
<tr>
<td>UNION relaxations</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>EXCEPT</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>INTERSECT</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>CORRESPONDING</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td><strong>Schema definition statement</strong></td>
<td>SQL92I</td>
<td></td>
</tr>
<tr>
<td>CREATE SCHEMA</td>
<td></td>
<td>yes, partially</td>
</tr>
<tr>
<td><strong>User authorization</strong></td>
<td>SQL92I</td>
<td></td>
</tr>
<tr>
<td>SET SESSION AUTHORIZATION</td>
<td></td>
<td>use set schema</td>
</tr>
<tr>
<td>CURRENT_USER</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>SESSION_USER</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>SYSTEM_USER</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td><strong>Constraint tables</strong></td>
<td>SQL92I</td>
<td></td>
</tr>
<tr>
<td>TABLE_CONSTRAINTS</td>
<td></td>
<td>SYS.SYSCONSTRAINTS</td>
</tr>
<tr>
<td>REFERENTIAL_CONSTRAINTS</td>
<td></td>
<td>SYS.SYSFOREIGNKEYS</td>
</tr>
<tr>
<td>CHECK_CONSTRAINTS</td>
<td></td>
<td>SYS.SYSCHECKS</td>
</tr>
</tbody>
</table>

*Cloudscape Version 3.0*
### Table 3-1  Support for SQL-92 Features (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage tables</td>
<td>SQL92I</td>
<td>SYS.SYSDEPENDS</td>
</tr>
<tr>
<td>Intermediate information schema</td>
<td>SQL92I</td>
<td>use JDBC</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>DatabaseMetaData</em> and <em>Cloudscape system tables</em></td>
</tr>
<tr>
<td>Subprogram support</td>
<td>SQL92I</td>
<td>not relevant to JDBC, which is much richer</td>
</tr>
<tr>
<td>Intermediate SQL Flagging</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>Schema manipulation</td>
<td>SQL92I</td>
<td>ADD and DROP CONSTRAINT, MODIFY DEFAULT</td>
</tr>
<tr>
<td>Long identifiers</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>Full outer join</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>Time zone specification</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>National character</td>
<td>SQL92I</td>
<td>support Unicode escapes and store Unicode strings</td>
</tr>
<tr>
<td>Scrolled cursors</td>
<td>SQL92I</td>
<td>partial (scrolling insensitive result sets through JDBC 2.0)</td>
</tr>
<tr>
<td>Intermediate set function support</td>
<td>SQL92I</td>
<td>partial</td>
</tr>
<tr>
<td>Character set definition</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>Named character sets</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>Scalar subquery values</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>Expanded null predicate</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>Constraint management</td>
<td>SQL92I</td>
<td>yes (ADD/DROP CONSTRAINT)</td>
</tr>
<tr>
<td><strong>Documentation schema</strong></td>
<td>SQL92I/FIPS 127-2</td>
<td>use JDBC</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>DatabaseMetaData</em></td>
</tr>
<tr>
<td>SQL_FEATURES</td>
<td>SQL92I</td>
<td>use JDBC</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>DatabaseMetaData</em></td>
</tr>
<tr>
<td>SQL_SIZING</td>
<td>SQL92I</td>
<td>use JDBC</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>DatabaseMetaData</em></td>
</tr>
<tr>
<td>BIT data types</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Assertion constraints</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Temporary tables</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Full dynamic SQL</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td><strong>Full DATETIME</strong></td>
<td>SQL92F</td>
<td></td>
</tr>
<tr>
<td>precision for TIME and TIMESTAMP</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Feature</td>
<td>Source</td>
<td>Cloudscape 3.0</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------</td>
<td>----------------</td>
</tr>
<tr>
<td>Full value expressions</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Truth value tests</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Full character functions</strong></td>
<td>SQL92F</td>
<td></td>
</tr>
<tr>
<td>POSITION expression</td>
<td></td>
<td>use Java methods</td>
</tr>
<tr>
<td>UPPER/LOWER functions</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Derived tables in FROM</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Trailing underscore</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Indicator data types</td>
<td>SQL92F</td>
<td>not relevant to JDBC</td>
</tr>
<tr>
<td>Referential name order</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Full SQL Flagging</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Row and table constructors</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Catalog name qualifiers</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Simple tables</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Subqueries in CHECK</td>
<td>SQL92F</td>
<td>no, but can do with Java methods</td>
</tr>
<tr>
<td>Union join</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Cross join</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Collation and translation</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Referential update actions</td>
<td>SQL92F</td>
<td>no, but can do with triggers</td>
</tr>
<tr>
<td>ALTER domain</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Deferable constraints</td>
<td>SQL92F</td>
<td>SET CONSTRAINTS</td>
</tr>
<tr>
<td>INSERT column privileges</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Referential MATCH types</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>View CHECK enhancements</td>
<td>SQL92F</td>
<td>no, views not updateable</td>
</tr>
<tr>
<td>Session management</td>
<td>SQL92F</td>
<td>use JDBC</td>
</tr>
<tr>
<td>Connection management</td>
<td>SQL92F</td>
<td>use JDBC</td>
</tr>
<tr>
<td>Self-referencing operations</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Insensitive cursors</td>
<td>SQL92F</td>
<td>Yes through JDBC 2.0</td>
</tr>
<tr>
<td>Full set function</td>
<td>SQL92F</td>
<td>partially</td>
</tr>
<tr>
<td>Catalog flagging</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Local table references</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Full cursor update</td>
<td>SQL92F</td>
<td>no</td>
</tr>
</tbody>
</table>
Cloudscape includes the following system tables:

- “SYSALIASES” on page 4-3
- “SYSCHECKS” on page 4-5
- “SYSCOLUMNS” on page 4-6
- “SYSCONGLOMERATES” on page 4-9
- “SYSCONSTRAINTS” on page 4-11
- “SYSDEPENDS” on page 4-13
- “SYSERRORS” on page 4-15
- “SYSFILES” on page 4-17
- “SYSFOREIGNKEYS” on page 4-18
- “SYSJDBCTYPEINFO” on page 4-19
- “SYSKEYS” on page 4-20
- “SYSSCHEMAS” on page 4-21
- “SYSSTATEMENTS” on page 4-22
- “SYSTABLES” on page 4-23
- “SYSTRIGGERS” on page 4-24
- “SYSVIEWS” on page 4-26

**NEW:** SYSMETHODALIASES has been eliminated in Version 3.0. It is replaced by the new table SYSALIASES.

You can query system tables, but you cannot alter them.

All system tables reside in the SYS schema. Because this is not the default schema, qualify all queries accessing the system tables with the SYS schema name.
You can also use an instance of the Java interface `java.sql.DatabaseMetaData` to get more information about these tables.

Some system tables store Java objects, so to get information from those tables you will need to examine the Java class that the column is defined to serialize. If a column serializes a Java class or interface, the class name or interface name is a link to the javadoc for that class. For more information, see “Using the Cloudscape Types to Query the System Tables” on page 4-17 in the Cloudscape Developer’s Guide.

**NOTE:** The HTML version of this document includes an entity-relationship diagram of these system tables.
**SYSALIASES**

Describes the method aliases, work units, user defined aggregates, and class aliases in the database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIASID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the alias</td>
</tr>
<tr>
<td>ALIAS</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>alias</td>
</tr>
<tr>
<td>JAVACLASSNAME</td>
<td>LONGVARCHAR</td>
<td>255</td>
<td>false</td>
<td>the Java class name</td>
</tr>
<tr>
<td>ALIASTYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>'A' (user aggregate)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'C' (class alias)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'M' (method alias)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'W' (work unit)</td>
</tr>
<tr>
<td>NAMESPACE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>'C' (class alias)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'M' (method alias)</td>
</tr>
<tr>
<td>SYSTEMALIAS</td>
<td>BOOLEAN</td>
<td></td>
<td>false</td>
<td>true (system supplied or built-in alias)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>false (alias created by a user)</td>
</tr>
<tr>
<td>ALIASINFO</td>
<td>SERIALIZABLE</td>
<td></td>
<td>true</td>
<td>A java interface that encapsulates the additional information that is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>specific to an alias</td>
</tr>
</tbody>
</table>

**NEW:** *SYSALIASES* is a new table in Version 3.0 and replaces *SYSMETHODALIASES*.

**Indexes**

- *SYSALIASES_INDEX1*, unique btree index on (ALIAS, NAMESPACE)
- *SYSALIASES_INDEX2*, unique btree index on (ALIASID)

**Useful Queries**

The query for returning all of the class aliases is:

```
SELECT ALIASID, ALIAS, JAVACLASSNAME
FROM SYS.SYSALIASES WHERE ALIASTYPE = 'C'
```

The query for returning all the built-in aliases is:

```
SELECT ALIASID, ALIAS, JAVACLASSNAME
FROM SYS.SYSALIASES WHERE SYSTEMALIAS = true
```
The query for returning all the method aliases is:

```
SELECT ALIASID, ALIAS, JAVACLASSNAME, ALIASINFO.getMethodName()
FROM SYS.SYSALIASES WHERE ALIASTYPE = 'M'
```

The query for returning all the user aggregates is:

```
SELECT ALIASID, ALIAS, JAVACLASSNAME
FROM SYS.SYSALIASES WHERE ALIASTYPE = 'A'
```

The query for returning all the work units is:

```
SELECT ALIASID, ALIAS, JAVACLASSNAME,
ALIASINFO.getMethodName(), ALIASINFO.getTargetClassName(),
ALIASINFO.getTargetMethodName()
FROM SYS.SYSALIASES WHERE ALIASTYPE = 'W'
```
SYSCHECKS

Describes the check constraints within the current database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the constraint</td>
</tr>
<tr>
<td>CHECKDEFINITION</td>
<td>LONG VARCHAR</td>
<td></td>
<td>false</td>
<td>text of check constraint definition</td>
</tr>
<tr>
<td>REFERENCEDCOLUMNS</td>
<td>SERIALIZE(COM.cloudscape.types.ReferencedColumnsDescriptor)</td>
<td></td>
<td>false</td>
<td>description of the columns referenced by the check constraint</td>
</tr>
</tbody>
</table>

**Indexes**

SYSCHECKS_INDEX1 unique BTREE index on (CONSTRAINTID)

**Example Query**

The following query returns the check constraints and the columns that they reference on table Flights:

```sql
SELECT CONSTRAINTNAME, COLUMNNAME
FROM SYS.SYSTABLES t, SYS.SYSCOLUMNS col,
     SYS.SYSCONSTRAINTS cons, SYS.SYSCHECKS checks
WHERE t.TABLENAME = 'FLIGHTS'
AND t.TABLEID = col.REFERENCEID
AND t.TABLEID = cons.TABLEID
AND cons.CONSTRAINTID = checks.CONSTRAINTID
AND REFERENCEDCOLUMNS.isReferencedColumn(col.COLUMNNUMBER)
ORDER BY CONSTRAINTNAME
```
Cloudscape System Tables

**SYSCOLUMNS**

Describes the columns within:

- all tables in the current database
- parameters in the current database’s publications
- parameters in the current database’s stored prepared statements

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFERENCENID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>Identifier for table, publication, or stored prepared statement (join with SYSTABLES.TABLEID, SYSSSTATEMENTS.STMTID, or SYSPUBS.PUBLICATIONID)</td>
</tr>
<tr>
<td>COLUMNNAME</td>
<td>CHAR</td>
<td>128</td>
<td>false</td>
<td>column or parameter name; null for stored prepared statements</td>
</tr>
<tr>
<td>COLUMNNUMBER</td>
<td>INT</td>
<td>4</td>
<td>false</td>
<td>the position of the column within the table or the position of the parameter within the publication or stored prepared statement</td>
</tr>
<tr>
<td>COLUMNDATATYPE</td>
<td>SERIALIZE (COM.cloudscape.types.TypeDescriptor)</td>
<td>false</td>
<td></td>
<td>system type that describes precision, length, scale, nullability, type name, and storage type of data</td>
</tr>
</tbody>
</table>
### NEW

The COLUMNDEFAULTID is a new column in `SYSCOLUMNS` in Version 3.0. The use of the COLUMNDEFAULT column is new (previously this column always stored NULL).

### Indexes

- `SYSCOLUMNS_INDEX1` unique BTREE index on `(REFERENCEID, COLUMNNAME)`
- `SYSCOLUMNS_INDEX2` unique BTREE index on `COLUMNDEFAULTID`

### Example Query

The following query returns some information about all the columns in the `Countries` table:
SELECT c.COLUMNNAME, c.COLUMNNUMBER, 
c.COLUMNDATATYPE.getSQLstring(), 
c.COLUMNDATATYPE.isNullable() 
FROM SYS.SYSCOLUMNS c, SYS.SYSTABLES t 
WHERE c.REFERENCEID = t.TABLEID 
AND t.TABLENAME = 'COUNTRIES'

-- see all columns that have default values
SELECT c.COLUMNNAME, COLUMNDEFAULT.toString(), TABLENAME 
FROM SYS.SYSCOLUMNS c, SYS.SYSTABLES s 
WHERE c.REFERENCEID=s.TABLEID 
AND COLUMNDEFAULT IS NOT NULL

-- see the default values for a stored prepared statement
SELECT STMTNAME, COLUMNDEFAULT.toString() 
FROM SYS.SYSCOLUMNS c, SYS.SYSTABLESTATEMENTS s 
WHERE c.REFERENCEID = s.STMTID 
AND STMTNAME = 'GETFLIGHTINFO'
SYSCONGLOMERATES

Describes the conglomerates within the current database. A conglomerate is a unit of storage and is either a table or an index.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>schema id for the conglomerate</td>
</tr>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>identifier for table (join with SYSTABLES.TABLEID)</td>
</tr>
<tr>
<td>CONGLOMERA TENUMBER</td>
<td>LONGINT</td>
<td>8</td>
<td>false</td>
<td>conglomerate id for the conglomerate (heap or index)</td>
</tr>
<tr>
<td>CONGLOMERA TENAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>true</td>
<td>index name, if conglomerate is an index, otherwise the table ID</td>
</tr>
<tr>
<td>ISINDEX</td>
<td>BOOLEAN</td>
<td>1</td>
<td>false</td>
<td>whether or not conglomerate is an index</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>SERIALIZE</td>
<td></td>
<td>true</td>
<td>system type describing the index</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>(COM.cloudscape.types.IndexDescriptor)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISCONSTRAINT</td>
<td>BOOLEAN</td>
<td>1</td>
<td>false</td>
<td>whether or not conglomerate is a system-generated index enforcing a constraint</td>
</tr>
<tr>
<td>CONGLOMERA TED</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the conglomerate</td>
</tr>
</tbody>
</table>

Indexes

- SYSCONGLOMERATES_INDEX1 unique BTREE index on (CONGLOMERATEID)
- SYSCONGLOMERATES_INDEX2 unique BTREE index on (CONGLOMERA TENAME, SCHEMAID)
- SYSCONGLOMERATES_INDEX3 BTREE index on (TABLEID)
Example Queries

The following query shows the key columns, their ordinal positions, and whether they are ascending or descending for an index named `new_index` in the table `Flights`, in the schema `APP`, in order:

```sql
SELECT
    CONGLOMS.DESCRIPTOR.getKeyColumnPosition(COLS.COLUMNNUMBER) AS ORDINAL_POSITION,
    COLS.COLUMNNAME AS COLUMN_NAME,
    (CONGLOMS.DESCRIPTOR.isAscending(CONGLOMS.DESCRIPTOR.getKeyColumnPosition(COLS.COLUMNNUMBER)) ? 'A' : 'D') AS ASC_OR_DESC
FROM SYS.SYSTABLES AS T,
     SYS.SYSCONGLOMERATES AS CONGLOMS,
     SYS.SYSCOLUMNS AS COLS,
     SYS.SYSSCHEMAS AS S
WHERE T.TABLEID = CONGLOMS.TABLEID
AND T.TABLEID = COLS.REFERENCEID
AND CONGLOMS.CONGLOMERATENAME = 'NEW_INDEX'
AND CONGLOMS.ISINDEX
AND S.SCHEMAID = T.SCHEMAID
AND S.SCHEMANAME = 'APP'
AND (CONGLOMS.DESCRIPTOR IS NOT NULL ? CONGLOMS.DESCRIPTOR.getKeyColumnPosition(COLS.COLUMNNUMBER) : 0) <> 0
ORDER BY ORDINAL_POSITION
```
**SYSCONSTRAINTS**

Describes the information common to all types of constraints within the current database (currently, this includes primary key, unique, foreign key, and check constraints).

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for constraint</td>
</tr>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>identifier for table (join with SYSTABLES.TABLEID)</td>
</tr>
<tr>
<td>CONSTRAINTNAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>constraint name (internally generated if not specified by user)</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>P (primary key), U (unique), C (check), or F (foreign key)</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>identifier for schema that the constraint belongs to (join with SYSSCHEMAS.SCHEMAID)</td>
</tr>
<tr>
<td>STATE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>E for enabled, D for disabled</td>
</tr>
<tr>
<td>REFERENCECOUNT</td>
<td>INTEGER</td>
<td>1</td>
<td>false</td>
<td>the count of the number of foreign key constraints that reference this constraint; this number can be greater than zero only for PRIMARY KEY and UNIQUE constraints</td>
</tr>
</tbody>
</table>

**Indexes**

- **SYSCONSTRAINTS_INDEX1** unique BTREE index on (CONSTRAINTID)
- **SYSCONSTRAINTS_INDEX2** unique BTREE index on (CONSTRAINTNAME, SCHEMAID)
- **SYSCONSTRAINTS_INDEX3** BTREE index on (TABLEID)

**Sample Queries**

Find out the names of all the primary key constraints in the database and the names of the indexes backing up those constraints:

```
SELECT c.CONSTRAINTNAME, cn.CONGLOMERATENAME
FROM SYS.SYSCONSTRAINTS c, SYS.SYSCONGLOMERATES cn, SYS.SYSKEYS k
WHERE c.TYPE = 'P'
AND c.CONSTRAINTID = k.CONSTRAINTID
AND cn.CONGLOMERATEID = k.CONGLOMERATEID
AND c.TABLEID = cn.TABLEID
```
Find out the name of the index backing up the constraint called `flights_pk` in the `Flights` table:

```sql
SELECT cn.CONGLOMERATENAME FROM SYS.SYSCONSTRAINTS c,
     SYS.SYSCONGLOMERATES cn, SYS.SYSKEYS k
WHERE c.CONSTRAINTID = k.CONSTRAINTID
AND cn.CONGLOMERATEID = k.CONGLOMERATEID
AND c.TABLEID = cn.TABLEID
AND c.CONSTRAINTNAME = 'FLIGHTS_PK'
```

Find out the foreign keys and the names of the backing indexes used to enforce them:

```sql
SELECT C.CONSTRAINTNAME, CN.CONGLOMERATENAME
FROM SYS.SYSCONSTRAINTS c, SYS.SYSFOREIGNKEYS f,
     SYS.SYSCONGLOMERATES cn
WHERE f.CONSTRAINTID = c.CONSTRAINTID
AND f.CONGLOMERATEID = cn.CONGLOMERATEID
```
SYSDEPENDS

Describes the dependency relationships between persistent objects in the database. Persistent objects can be dependents (they depend on other objects) and/or providers (other objects depend on them). When the user attempts to drop or modify a provider, Cloudscape checks to see if there are any current dependents. If there are, one of the following actions will occur:

- If the dependent is not a stored prepared statement, an exception is thrown saying that the dependents must be dropped first before dropping the provider.
- If the dependent is a stored prepared statement, it will be marked as needing recompilation.

Providers are tables, conglomerates, and constraints. Dependents are publications, views, and stored prepared statements.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPENDENTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the dependent</td>
</tr>
<tr>
<td>DEPENDENTFINDER</td>
<td>SERIALIZE (COM.cloudscape.types. DependableFinder)</td>
<td>1</td>
<td>false</td>
<td>system type describing the publication, view, or stored prepared statement</td>
</tr>
<tr>
<td>PROVIDERID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the provider</td>
</tr>
<tr>
<td>PROVIDERFINDER</td>
<td>SERIALIZE (COM.cloudscape.types. DependableFinder)</td>
<td>1</td>
<td>false</td>
<td>system type describing the tables, conglomerates, and constraints that are providers</td>
</tr>
</tbody>
</table>

Indexes

- SYSDEPENDS_INDEX1 BTREE index on (DEPENDENTID)
- SYSDEPENDS_INDEX2 BTREE index on (PROVIDERID)

The following query returns the dependent object names and types and the provider names and types that they are dependent on:

```sql
SELECT dependentFinder.getSQLObjectName(dependentid) AS DEPENDENT_OBJECT_NAME,
       DEPENDENTFINDER.sqlObjectType() AS DEPENDENT_OBJECT_TYPE,
       providerFinder.getSQLObjectName(providerid) AS PROVIDER_OBJECT_NAME,
       PROVIDERFINDER.sqlObjectType() AS PROVIDER_OBJECT_TYPE
FROM SYSDEPENDS
```
The following query returns all of the providers on which a view (Segments_Seatbookings) is dependent:

```
SELECT PROVIDERFINDER.getSQLObjectName(providerid) AS PROVIDER_OBJECT_NAME,
       PROVIDERFINDER.getSQLObjectType() as PROVIDER_OBJECT_TYPE
FROM SYS.SYSDEPENDS
WHERE dependentFinder.getSQLObjectName(dependentid) =
       'SEGMENTS_SEATBOOKINGS'
```

**Marking Dependent Stored Prepared Statements Invalid**

Stored prepared statements are “dependents”; that is, they depend on other dictionary objects, which are their “providers.” When a provider is changed or dropped, Cloudscape marks the stored prepared statement invalid, thus requiring recompilation at the next execution.

The invalidated target stored prepared statement is not recompiled until the next time the statement is executed.

Each time a statement is recompiled (either deliberately or as a side effect of some other action), the LASTCOMPILED field of SYS.SYSSTATEMENTS is updated with the timestamp at the time of recompilation.
**SYSERRORS**

This table is created in all databases. In a synchronization system, this table holds descriptions of consistency errors that occur in replicating user transactions. In a non-synchronization system, this table is empty.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERRORID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the error</td>
</tr>
<tr>
<td>ERRORTIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>false</td>
<td>time error occurred</td>
</tr>
<tr>
<td>TRANSACTIONDBID</td>
<td>CHAR</td>
<td>36</td>
<td>true</td>
<td>ID of database in which transaction failed</td>
</tr>
<tr>
<td>ERRORINFO</td>
<td>SERIALIZE (&lt;COM.cloudscape.types.ErrorInfo&gt;)</td>
<td>false</td>
<td>description of the error</td>
<td></td>
</tr>
<tr>
<td>PUBLICATIONID</td>
<td>CHAR</td>
<td>36</td>
<td>true</td>
<td>The publication ID for a transaction that failed when being replayed</td>
</tr>
<tr>
<td>STMTCOUNTERS</td>
<td>SERIALIZE(int[])</td>
<td>true</td>
<td>In the usual case, the array has on element, which holds the statement number (inside the transaction) where the error occurred. In the case of nested statements (statements inside statements), there may be many cells in the array. From left to right, the cells hold the statement number (from outermost to innermost) where the error occurred.</td>
<td></td>
</tr>
<tr>
<td>TRANCONTENTS</td>
<td>LONG BIT VARYING</td>
<td>true</td>
<td>Holds a bit stream, which is the serialized list of replicated statements in this transaction.</td>
<td></td>
</tr>
</tbody>
</table>

**NEW:** The columns `PUBLICATIONID`, `STMTCOUNTERS`, and `TRANCONTENTS` are new in `SYSERRORS` in Version 3.0.

**Indexes**

- `SYSERRORS_INDEX1` unique BTREE index on `(ERRORID)`
- `SYSERRORS_INDEX2` BTREE index on `(ERRORTIME)`
• SYSERRORS_INDEX3 unique BTREE index on (TRANSACTIONDBID)
SYSFILES

Describes jar files stored in the database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the jar file</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>ID of the jar file’s schema (join with SYSSCHEMAS.SCHEMAID)</td>
</tr>
<tr>
<td>FILENAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>SQL name of the jar file</td>
</tr>
<tr>
<td>GENERATIONID</td>
<td>LONGINT</td>
<td></td>
<td>false</td>
<td>Generation number for the file. When jar files are replaced, their generation identifiers are changed.</td>
</tr>
</tbody>
</table>

**Indexes**

- `SYSFILES_INDEX1` unique BTREE index on `(FILENAME, SCHEMAID)`
- `SYSFILES_INDEX2` unique BTREE index on `(FILEID)`
Cloudscape System Tables

SYSFOREIGNKEYS

Describes the information specific to foreign key constraints in the current database.

Cloudscape generates a backing index for each foreign key constraint; the name of this index is the same as SYSFOREIGNKEYS.CONGLOMERATEID.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the foreign key constraint (join with SYSCONSTRAINTS.CONSTRAINTID)</td>
</tr>
<tr>
<td>CONGLOMERATEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for index backing up the foreign key constraint (join with SYSCONGLOMERATES.CONGLOMERATEID)</td>
</tr>
<tr>
<td>KEYCONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the primary key or unique constraint referenced by this foreign key (SYSKEYS.CONSTRAINTID or SYSCONSTRAINTS.CONSTRAINTID)</td>
</tr>
<tr>
<td>DELETERULE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>R for restrict</td>
</tr>
<tr>
<td>UPDATERULE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>R for restrict</td>
</tr>
</tbody>
</table>

Indexes

- SYSFOREIGNKEYS_INDEX1 unique BTREE index on (CONSTRAINTID)
- SYSFOREIGNKEYS_INDEX2 BTREE index on (KEYCONSTRAINTID)
SYSJDBCTYPEINFO

Used internally for the JDBC DatabaseMetaData methods. One row per type is built into the system. (See JDBC Database Access with Java, Hamilton, Cattell & Fisher, p. 209, for column descriptions.) The column-naming style differs from that of other system tables to conform to the JDBC DatabaseMetaData ResultSet column names.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_NAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
</tr>
<tr>
<td>DATA_TYPE</td>
<td>SMALLINT</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>PRECISION</td>
<td>INT</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>LITERAL_PREFIX</td>
<td>VARCHAR</td>
<td>10</td>
<td>true</td>
</tr>
<tr>
<td>LITERAL_SUFFIX</td>
<td>VARCHAR</td>
<td>10</td>
<td>true</td>
</tr>
<tr>
<td>CREATE_PARAMS</td>
<td>VARCHAR</td>
<td>10</td>
<td>true</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>INT</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>CASE_SENSITIVE</td>
<td>BOOLEAN</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>SEARCHABLE</td>
<td>INT</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>UNSIGNED_ATTRIBUTE</td>
<td>BOOLEAN</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>FIXED_PREC_SCALE</td>
<td>BOOLEAN</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>AUTO_INCREMENT</td>
<td>BOOLEAN</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>LOCAL_TYPE_NAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
</tr>
<tr>
<td>MINIMUM_SCALE</td>
<td>SMALLINT</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>MAXIMUM_SCALE</td>
<td>SMALLINT</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>SQL_DATA_TYPE</td>
<td>INT</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>SQL_DATETIME_SUB</td>
<td>INT</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>NUM_PREC_RADIX</td>
<td>INT</td>
<td>4</td>
<td>false</td>
</tr>
</tbody>
</table>
SYSKEYS

Describes the specific information for primary key and unique constraints within the current database. Cloudscape generates an index on the table to back up each such constraint. The index name is the same as $SYSKEYS.CONGLOMERATEID$.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for constraint</td>
</tr>
<tr>
<td>CONGLOMEREATEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for backing index</td>
</tr>
</tbody>
</table>

Indexes

$SYSKEYS_INDEX1$ unique BTREE index on ($CONSTRAINTID$)

Example Queries

The following query returns the name of the index backing up the primary key constraint called $FlightAvailability_{pk}$:

```
SELECT cn.CONGLOMEREATENAME FROM SYS.SYSCONSTRAINTS c,
    SYS.SYSKEYS k, SYS.SYSCONGLOMERATES cn
WHERE cCONSTRAINTID = kCONSTRAINTID
AND kCONGLOMEREATEID = cnCONGLOMEREATEID
AND cCONSTRAINTNAME = 'FLIGHTAVAILABILITY_PK'
```
SYSSCHEMAS

Describes the schemas within the current database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the schema</td>
</tr>
<tr>
<td>SCHEMANAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>schema name</td>
</tr>
<tr>
<td>AUTHORIZATIONID</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>the authorization identifier of the owner of the schema</td>
</tr>
</tbody>
</table>

**Indexes**

- `SYSSCHEMAS_INDEX1` unique BTREE index on `(SCHEMANAME)`
- `SYSSCHEMAS_INDEX2` unique BTREE index on `(SCHEMAID)`
**SYSSTATEMENTS**

Contains one row per stored prepared statement. Includes rows for SQL-J text is a trigger action.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>STMTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the statement</td>
</tr>
<tr>
<td>STMTNAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>name of the statement</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>the schema in which the statement resides</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>always 'S'</td>
</tr>
<tr>
<td>VALID</td>
<td>BOOLEAN</td>
<td></td>
<td>false</td>
<td>TRUE if valid, FALSE if invalid</td>
</tr>
<tr>
<td>TEXT</td>
<td>LONG VARCHAR</td>
<td></td>
<td>false</td>
<td>text of the statement</td>
</tr>
<tr>
<td>LASTCOMPILED</td>
<td>TIMESTAMP</td>
<td></td>
<td>false</td>
<td>time that the statement was compiled</td>
</tr>
<tr>
<td>COMPILETIONSCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the schema containing the statement</td>
</tr>
<tr>
<td>USINGTEXT</td>
<td>LONG VARCHAR</td>
<td></td>
<td>true</td>
<td>text of the USING clause of the CREATE STATEMENT and ALTER STATEMENT statements</td>
</tr>
</tbody>
</table>

**Indexes**

- **SYSSTATEMENTS_INDEX1** unique BTREE index on **STMTID**
- **SYSSTATEMENTS_INDEX2** unique BTREE index on **STMTNAME**, **SCHEMAID**

**Sample Queries**

```sql
-- Find information about each of the parameters used by
-- the getFullFlightInfo statement. Return the position
-- of the parameter, its sample value (if any), and its type
SELECT COLUMNNUMBER, COLUMNDEFAULT, COLUMNDATATYPE.toString()
FROM SYS.SYSCOLUMNS C, SYS.SYSSTATEMENTS S
WHERE C.REFERENCEID = S.STMTID
AND S.STMTNAME = 'GETFULLFLIGHTINFO'
```
SYSTABLES

Describes the tables and views within the current database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for table or view</td>
</tr>
<tr>
<td>TABLENAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>table or view name</td>
</tr>
<tr>
<td>TABLETYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘S’ (system table), ‘T’ (user table), or ‘V’ (view)</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>schema id for the table or view</td>
</tr>
<tr>
<td>LOCKGRANULARITY</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>Indicates the lock granularity for the table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>‘T’ (table level locking)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>‘R’ (row level locking, the default)</td>
</tr>
</tbody>
</table>

**NEW:** The `LOCKGRANULARITY` column is a new column in `SYSTABLES` in Version 3.0.

**Indexes**

- `SYSTABLES_INDEX1` unique BTREE index on (TABLENAME, SCHEMAID)
- `SYSTABLES_INDEX2` unique BTREE index on (TABLEID)
SYSTRIGGERS

Describes the database’s triggers.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIGGERID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the trigger</td>
</tr>
<tr>
<td>TRIGGERNAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>name of the trigger</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the trigger’s schema (join with SYSSCHEMAS.SCHEMAID)</td>
</tr>
<tr>
<td>CREATIONTIMESTAMP</td>
<td>TIMESTAMP</td>
<td></td>
<td>false</td>
<td>time the trigger was created</td>
</tr>
<tr>
<td>EVENT</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘U’ for update, ‘D’ for delete, ‘I’ for insert</td>
</tr>
<tr>
<td>FIRINGTIME</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘B’ for before ‘A’ for after</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘R’ for row, ‘S’ for statement</td>
</tr>
<tr>
<td>STATE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘E’ for enabled, ‘D’ for disabled</td>
</tr>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the table on which the trigger is defined</td>
</tr>
<tr>
<td>REFERENCEDCOLUMNS</td>
<td>SERIALIZE(</td>
<td>true</td>
<td></td>
<td>descriptor of the columns referenced by UPDATE triggers</td>
</tr>
<tr>
<td></td>
<td>COM.cloudscape.</td>
<td></td>
<td></td>
<td>types. ReferencedColumns Descriptor)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHENSTMTID</td>
<td>CHAR</td>
<td>36</td>
<td>true</td>
<td>used only if there is a WHEN clause (not yet supported)</td>
</tr>
<tr>
<td>ACTIONSTMTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the stored prepared statement for the trigger action (join with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SYSSSTATEMENTS.STMTID)</td>
</tr>
</tbody>
</table>

Any SQL-J text that is part of a trigger action is compiled and stored in SYSSSTATEMENTS. ACTIONSTMTID and WHENSTMTID are foreign keys that reference SYSSSTATEMENTS.STMTID. The statements for a trigger are always in the same schema as the trigger.

**NEW:** SYSTRIGGERS is new in Version 3.0.
Indexes

- **SYSTRIGGERS_INDEX1** unique BTREE index on (TRIGGERID)
- **SYSTRIGGERS_INDEX2** unique BTREE index on (TRIGGERNAME, SCHEMAID)
- **SYSTRIGGERS_INDEX3** BTREE index on (TABLEID, CREATIONTIMESTAMP)

-- get the name, event, and type of each trigger
SELECT t.TRIGGERNAME, t.EVENT,
       (t.TYPE='R'?'ROW': 'STATEMENT') AS TYPE
FROM SYS.SYSTRIGGERS t, SYS.SYSTABLES tbl
WHERE t.tableID = tbl.tableId

-- get the text action for each trigger
SELECT t.TRIGGERNAME, s.TEXT
FROM SYS.SYSTRIGGERS t, SYS.SYSSTATEMENTS s
WHERE t.ACTIONSTMTID = s.STMTID
**SYSVIEWS**

Describes the view definitions within the current database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the view (called TABLEID since it is joined with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>column of that name in SYSTABLES)</td>
</tr>
<tr>
<td>VIEWDEFINITION</td>
<td>LONG VARCHAR</td>
<td>false</td>
<td></td>
<td>text of view definition</td>
</tr>
<tr>
<td>CHECKOPTION</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘N’ (check option not supported yet)</td>
</tr>
<tr>
<td>COMPILATIONSCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the schema containing the view</td>
</tr>
</tbody>
</table>

**Indexes**

SYSVIEWS_INDEX1 unique BTREE index on (TABLEID)
The JDBC driver returns SQLExceptions for all errors from Cloudscape. If the exception originated in a user type but is not itself an SQLException, it is wrapped in an SQLException. Cloudscape-specific SQLExceptions use SQLState class codes starting with X. Standard SQLState values are returned for exceptions where appropriate.

Cloudscape database exceptions are classified by severity. The severity of an SQLException is available through the getErrorCode method call on the SQLException. The severities are summarized below. For more information, check the javadoc for COM.cloudscape.types.JBMSExceptionSeverity:

- Statement Severity—the effects of the current statement, if any, on persistent data are undone.
- Transaction Severity—the effects of the current transaction on persistent data are undone; a rollback is performed.
- Session Severity—a rollback is performed and the current session is terminated. This closes the current connection.
- System Severity—the system is shut down. All uncommitted transactions are rolled back.

Cloudscape provides a class, COM.cloudscape.database.JBMSException, for checking the severity of an exception.

Unimplemented aspects of the JDBC driver return an SQLException with a message starting “Feature not implemented” and an SQLState of XJZZZ. These unimplemented parts are for features not supported by Cloudscape.
Cloudscape supplies values for the message and SQLState fields. In addition, Cloudscape sometimes returns multiple SQLExceptions using the nextException chain. The first exception is always the most severe exception, with SQL-92 Standard exceptions preceding those that are specific to Cloudscape.

For information on processing SQLExceptions, see “Working with Cloudscape SQLExceptions in an Application” on page 6-34 in the Cloudscape Developer’s Guide.

SQLStates for some common errors:

- deadlock
  40001
- foreign key violation
  23501
- check constraint violation
  23L02
- duplicate value violating unique or primary key constraint
  23500
- duplicate value violating unique index
  23L01
- truncation error
  22001
- user authorization error
  XJ006
- user authentication error (no permission to access database)
  04501
- user authentication error (no write access)
  22502

NOTE: Cloudscape reserves the right to change the SQLState of Cloudscape errors.
Cloudscape comes with a built-in JDBC driver. That makes the JDBC API the main API for working with Cloudscape databases. The driver is a native protocol all-Java driver (type #4 among the categories defined by JavaSoft.)

This chapter provides reference information about Cloudscape’s implementation of the JDBC API and documents the way it conforms to the JDBC 1.2 API and the JDBC 2.0 API.

**NEW:** Beginning with Version 3.0, Cloudscape supports a subset of the new JDBC 2.0 features. See “JDBC 2.0-Only Features” on page 6-26.

See the *Cloudscape Developer’s Guide* for task-oriented instructions on working with the driver.

This JDBC driver implements the standard JDBC interface defined by JavaSoft. When invoked from an application running in the same JVM as Cloudscape, the JDBC driver supports connections to a Cloudscape database in local mode. No network transport is required to access the database. In client/server mode, the client application dispatches JDBC requests to the JDBC server over a network; the server, in turn, which runs in the same JVM as Cloudscape, sends requests to Cloudscape through the local JDBC driver.

The Cloudscape JDBC implementation provides access to Cloudscape databases and supplies all the required JDBC interfaces. Unimplemented aspects of the JDBC driver return an `SQLException` with a message stating “Feature not implemented” and an `SQLState` of XJZZZ. These unimplemented parts are for features not supported by Cloudscape.

- “Core JDK 1.1.x java.sql Classes, Interfaces, and Methods” on page 6-2
- “JDBC 2.0-Only Features” on page 6-26
- “JDBC Escape Syntax” on page 6-35
Core JDK 1.1.x java.sql Classes, Interfaces, and Methods

This section details the JDBC implementation of the following java.sql classes, interfaces, and methods:

- “java.sql.Driver” on page 6-3
- “java.sql.DriverManager.getConnection” on page 6-5
- “java.sql.DriverManager.getPropertyInfo” on page 6-10
- “java.sql.Connection” on page 6-11
- “java.sql.DatabaseMetaData” on page 6-12
- “java.sql.Statement” on page 6-14
- “java.sql.PreparedStatement” on page 6-15
- “java.sql.CallableStatement” on page 6-17
- “java.sql.ResultSet” on page 6-20
- “java.sql.ResultSetMetaData” on page 6-22
- “java.sql.SQLException” on page 6-23
- “java.sql.SQLWarning” on page 6-24
- “java.sql.Types” on page 6-25
JDBC provides different ways to load a driver, listed below.

- “Embedded Databases” on page 6-3
- “Cloudconnector” on page 6-4
- “RmiJdbc” on page 6-4

**Embedded Databases**

The class that loads Cloudscape’s local JDBC driver is the class

`COM.cloudscape.core.JDBCDriver`. Some of the ways listed below create
instances of the Cloudscape driver class. Do not use the class directly via the
`java.sql.Driver` interface. Use the `DriverManager` class to create connections.

- `Class.forName("COM.cloudscape.core.JDBCDriver")`
The manner recommended by JavaSoft. However, some JVMs may not
load the class when it is accessed. They can delay until instances of the
class are created.

For example, this method doesn’t work on AIX. Instead, use:

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

Adding the `newInstance()` method guarantees that the Cloudscape system
will be booted in all JVMs.

- `Class.forName("COM.cloudscape.core.JDBCDriver").newInstance()` Our recommended manner, because it ensures that the class is loaded in all
JVMs by creating an instance at the same time.

- `new COM.cloudscape.core.JDBCDriver()` Same as
`Class.forName("COM.cloudscape.core.JDBCDriver").newInstance()`, except that it requires the class to be found when the code is compiled.

- `Class c = COM.cloudscape.core.JDBCDriver.class`
This is the same as `Class.forName("COM.cloudscape.core.JDBCDriver")`, except that it requires the class to be found when the code is compiled. The
pseudo-static field `class` evaluates to the class that is named.

- **Setting the System property jdbc.drivers**
To set a System property, you alter the invocation command line or the
system properties within your application. It is not possible to alter system
properties within an applet.

An invocation of the JDK JVM:
The actual driver that gets registered in the DriverManager to handle the jdbc:cloudscape: protocol is not the class COM.cloudscape.core.JDBCDriver; that class simply detects the type of Cloudscape driver needed and then causes the appropriate Cloudscape driver to be loaded.

The only supported way to connect to a Cloudscape system through the jdbc:cloudscape: protocol is using the DriverManager to obtain a driver (java.sql.Driver) or connection (java.sql.Connection) through the getDriver and getConnection method calls.

Cloudconnector

For clients to Cloudconnector, you load the client JDBC driver with the class COM.cloudscape.core.WebLogicDriver in the same manner as that prescribed for an embedded environment. See the Cloudscape Server and Administration Guide for more information.

Rmijdbc

For clients to the Rmijdbc server, you load the client JDBC driver with the class COM.cloudscape.core.RmiJdbcDriver in the same manner as that prescribed for an embedded environment. See the Cloudscape Server and Administration Guide for more information.
java.sql.DriverManager.getConnection

A Java application using the JDBC API establishes a connection to a database by obtaining a `Connection` object. The standard way to obtain a `Connection` object is to call the method `DriverManager.getConnection`, which takes a String containing a database connection URL. A JDBC database connection URL (uniform resource locator) provides a way of identifying a database.

`DriverManager.getConnection` can take one argument besides a database connection URL, a `Properties` object. You can use the `Properties` object to set database connection URL attributes.

You can also supply strings representing user names and passwords. When they are supplied, Cloudscape checks whether they are valid for the current system if user authentication is enabled. User names are passed to Cloudscape as authorization identifiers, which are used to determine whether the user is authorized for access to the database and for determining the default schema. When the connection is established, if no user is supplied, Cloudscape sets the default user to `APP`, which Cloudscape uses to name the default schema. If a user is supplied, the default schema is the same as the user name if one exists (otherwise, it’s `APP`).

**Cloudscape Database Connection URL Syntax**

A Cloudscape database connection URL consists of the basic database connection URL followed by an optional subsubprotocol and optional attributes.

This section provides reference information only. For a more complete description, including examples, see “Connecting to Databases” on page 2-16 in the *Cloudscape Developer’s Guide*.

**Embedded Databases**

For applications with embedded databases, the syntax of the database connection URL is

```
jdbc:cloudscape:[subsubprotocol:][databaseName][;attributes]*
```

- `jdbc:cloudscape:`
  In JDBC lingo, `cloudscape` is the subprotocol for connecting to a Cloudscape database. The subprotocol is always `cloudscape` and does not vary.

- `subsubprotocol:`
  `subsubprotocol`, which is not typically specified, specifies where Cloudscape looks for a database: in a directory, in a class path, or in a jar.
file. It is used only in rare instances, usually for read-only databases.

subsubprotocol

is one of the following:

- directory
- classpath
  Databases are treated as read-only databases, and all databaseNames
  must begin with at least a slash, because you specify them “relative” to
  the class path directory or archive. (You do not have to specify classpath
  as the subsubprotocol; it is implied.)
- jar
  Databases are treated as read-only databases.

jar requires an additional element immediately before the databaseName:

(pathToArchive)

pathToArchive

is the path to the jar or zip file that holds the database and
includes the name of the jar or zip file.
See the Cloudscape Developer’s Guide for examples of database
connection URLs for read-only databases.

• databaseName
  Specify the databaseName to connect to an existing database or a new one.
  You can shut down the current database if you specify current=true for the
databaseName combined with the shutdown=true attribute.
  You can specify the database name alone, or with a relative or absolute
  path. See “Standard Connections—Connecting to Databases in the File
  System” on page 2-17 in the Cloudscape Developer’s Guide.

• attributes
  Specify 0 or more database connection URL attributes as detailed in “The
Cloudscape Database Connection URL Attributes” on page 6-9.

**Syntax—Clients to Cloudconnector**

For clients to Cloudconnector, the syntax of the database connection URL is

```
jdbc:cloudscape:weblogic[-ssl]://[hostname:portnum/]
[subsubprotocol:]\[databaseName]\;cloudscapeAttributes]
[\&webLogicAttributes]
```

Use -ssl after the word weblogic if you are using Cloudconnector with SSL turned
on.

• //hostname:portnum/
  Replace hostname with the name of the host running Cloudconnector if it
  is not running on the same machine as the client. Replace portnum with the
java.sql.DriverManager.getConnection

port number specified for the `weblogic.system.listenPort` property in the `weblogic.properties` file or on the Cloudconnector JVM command line. 7001 is the default value. 7002 is the default value if SSL is turned on.

If you do not specify `//hostname:portnum/`, the default value is `//localhost:7001/`. For weblogic-ssl, the default value is `//localhost:7002/`.

- **subsubprotocol**: subsubprotocol, which is not typically specified, specifies where Cloudscape looks for a database: in a directory, in a class path, or in a jar file. It is used only in rare instances, usually for read-only databases.

  subsubprotocol is one of the following:

  - directory
  - classpath
    Databases are treated as read-only databases, and all `databaseNames` must begin with at least a slash, because you specify them “relative” to the class path directory or archive.
  - jar
    Databases are treated as read-only databases.
  - jar
    Databases are treated as read-only databases.

  `jar` requires an additional element immediately before the `databaseName`:

  ```
  (pathToArchive)
  ```

  `pathToArchive` is the path to the jar or zip file that holds the database and includes the name of the jar or zip file.

- **`[databaseName][;cloudscapeAttributes]`**

  Specify the database name and any database connection URL attributes as specified in “The Cloudscape Database Connection URL Attributes” on page 6-9.

- **`webLogicAttributes`**

  You can set Cloudconnector (WebLogic) properties such as prefetching and caching. Properties set on a database connection URL are valid for the current session only. For examples, see the Cloudscape Server and Administration Guide.

  Separate each WebLogic property with a new ampersand (&).

**Syntax—Clients to RmiJdbc Server**

For clients to RmiJdbc Server, the syntax of the database connection URL is

```
jdbc:cloudscape:rmi://hostname:portnum/[subsubprotocol:] [databaseName][;cloudscapeAttributes]
```
NEW: This format of the RmiJdbc database connection URL is new in Version 3.0.

- //hostname:portnum/
  Replace hostname with the name of the host running Cloudconnector if it is not running on the same machine as the client. Replace portnum with the port number specified for the weblogic.system.listenPort property in the weblogic.properties file or on the Cloudconnector JVM command line. 1099 is the default value.
  If you do not specify //hostname:portnum/, the default value is //localhost:1099/.

- subsubprotocol:
  subsubprotocol, which is not typically specified, specifies where Cloudscape looks for a database: in a directory, in a class path, or in a jar file. It is used only in rare instances, usually for read-only databases.
  subsubprotocol is one of the following:
    - directory
    - classpath
      Databases are treated as read-only databases, and all databaseNames must begin with at least a slash, because you specify them “relative” to the class path directory or archive.
    - jar
      Databases are treated as read-only databases.
      jar requires an additional element immediately before the databaseName:

        (pathToArchive)

      pathToArchive is the path to the jar or zip file that holds the database and includes the name of the jar or zip file.

- [databaseName][;cloudscapeAttributes]
  Specify the database name and any database connection URL attributes as specified in “The Cloudscape Database Connection URL Attributes” on page 6-9.

Additional Syntax

Cloudscape also supports the following syntax:

    jdbc:default:connection

This database connection URL is equivalent to the standard Cloudscape using the current=true attribute. For more information, see “current=true” on page 7-5.
The Cloudscape Database Connection URL Attributes

You can supply an optional list of attributes to a database connection URL. Cloudscape translates these attributes into properties, so you can also set attributes in a Properties object passed to DriverManager.getConnection. (You cannot set those attributes as system properties, only in an object passed to the DriverManager.getConnection method.)

These attributes are specific to Cloudscape and are listed in Chapter 7, “Database Connection URL Attributes”.

Attribute name/value pairs are converted into properties and added to the properties provided in the connection call. If no properties are provided in the connection call, a properties set is created that contains only the properties obtained from the database connection URL.

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

-- setting an attribute in a Properties object
Properties p = new Properties();
p.put("create", "true");
Connection conn = DriverManager.getConnection(
    "jdbc:cloudscape:toursDB", myProps);

-- passing user name and password
Connection conn = DriverManager.getConnection(  
    "jdbc:cloudscape:toursDB", "dba", "password");
```

**NOTE:** Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored.
To get the \texttt{DriverPropertyInfo} object, request the JDBC driver from the driver manager:

\begin{verbatim}
java.sql.DriverManager.getDriver("jdbc:cloudscape:").
getPropertyInfo(URL, Prop)
\end{verbatim}

Do not request it from \texttt{COM.cloudscape.core.JDBCDriver}, which is only an intermediary class that loads the actual driver.

This method may return a \texttt{DriverPropertyInfo} object. In a Cloudscape system, it consists of an array of database connection URL attributes. The most useful attribute is limited to \texttt{databaseName=nameofDatabase}, which means that the object consists of a list of booted databases in the current system.

For example, if a Cloudscape system has the databases \texttt{toursDB} and \texttt{flightsDB} in its system directory, all the databases in the system are set to boot automatically, and a user has also connected to a database \texttt{A:/dbs/tours94}, the array returned from \texttt{getPropertyInfo} contains one object corresponding to the \texttt{databaseName} attribute. The choices field of the \texttt{DriverPropertyInfo} object will contain an array of three Strings with the values \texttt{toursDB}, \texttt{flightsDB}, and \texttt{A:/dbs/tours94}. Note that this object is returned only if the proposed connection objects do not already include a database name (in any form) or include the shutdown attribute with the value true.

For more information about \texttt{java.sql.Driver.getPropertyInfo}, see “Offering Connection Choices to the User” on page 9-2 in the Cloudscape Developer’s Guide.
A Cloudscape Connection object is not garbage-collected until all other JDBC objects created from that connection are explicitly closed or are themselves garbage-collected. Once the connection is closed, no further JDBC requests can be made against objects created from the connection. Do not explicitly close the Connection object until you no longer need it for executing statements.

A session-severity or higher exception causes the connection to close and all other JDBC objects against it to be closed. System-severity exceptions cause the Cloudscape system to shut down, which not only closes the connection but means that no new connections should be created in the current JVM.

Only java.sql.Connection.TRANSACTION_SERIALIZABLE and java.sql.Connection.TRANSACTION_READ_COMMITTED transaction isolations are available from a Cloudscape database; attempting to set isolation to another raises an SQLException.

TRANSACTION_READ_COMMITTED is the default isolation level.

Changing the current isolation for the connection with setConnection commits the current transaction and begins a new transaction, per the JDBC standard.

java.sql.Connection.setReadOnly

java.sql.Connection.setReadOnly is supported.

java.sql.Connection.isReadOnly

If you connect to a read-only database, the appropriate isReadOnly DatabaseMetaData value is returned. For example, Connections set to read-only using the setReadOnly method, Connections for which the user has been defined as a readOnlyAccess user (with one of the Cloudscape properties), and Connections to databases on read-only media return true.

Connection Functionality Not Supported

Cloudscape does not use catalog names; the getCatalog and setCatalog methods result in a “Feature not implemented” SQLException with an SQLState of XJZZZ.
java.sql.DatabaseMetaData

**Stored Prepared Statements for DatabaseMetaData Queries**

The JDBC driver has built-in queries that it uses to supply the results of 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, not Cloudscape-specific applications, which can query system catalogs directly.

The cloudscape.jdbc.metadataStoredPreparedStatements property configures Cloudscape’s built-in JDBC driver to take advantage of stored prepared statements to avoid preparing these queries each time an application starts up.

JDBC DatabaseMetaData queries are duplicated in every Cloudscape database even if a single Cloudscape system manages several databases.

With this property, you have the option of having Cloudscape create and store the statements as needed (dynamically) or all at once when the database is created.

For more information, see “cloudscape.jdbc.metadataStoredPreparedStatements” on page 5-29 in Tuning Cloudscape.

**NEW:** The getColumns method returns a ResultSet that contains a column called COLUMN_DEF. This method returns the default value of the columns, which are newly supported in Version 3.0.

**DatabaseMetaData Result Sets**

DatabaseMetaData result sets do not close the result sets of other statements, even when auto-commit is set to true.

DatabaseMetaData result sets are closed if a user performs any other action on a JDBC object that causes an automatic commit to occur. If you need the DatabaseMetaData result sets to be accessible while executing other actions that would cause automatic commits, turn off auto-commit with setAutoCommit(false).

**DatabaseMetaData Functionality Not Supported**

In the current release, Cloudscape does not provide all of the DatabaseMetaData functionality. The following JDBC requests result in empty result sets, in the format required by the JDBC API:

- getColumnPrivileges
- getProcedureColumns

*Cloudscape Reference Manual*
• `getTablePrivileges`

Cloudscape does not implement privileges, and thus has no information to provide for these calls.

`getBestRowIdentifier` looks for identifiers in this order:

• a primary key on the table
• a unique constraint or unique index on the table
• all the columns in the table

Because of this last choice, it will always find a set of columns that identify a row. However, if there are duplicate rows in the table, use of all columns may not necessarily identify a unique row in the table.
ResultSet Objects

An error that occurs when a SELECT statement is first executed prevents a ResultSet object from being opened on it. The same error does not close the ResultSet if it occurs after the ResultSet has been opened.

For example, a divide-by-zero error that happens while the executeQuery method is called on a java.sql.Statement or java.sql.PreparedStatement throws an exception and returns no result set at all, while if the same error happens while the next method is called on a ResultSet object, it does not cause the result set to be closed.

Errors can happen when a ResultSet is first being created if the system partially executes the query before the first row is fetched. This can happen on any query that uses more than one table and on queries that use aggregates, GROUP BY, ORDER BY, DISTINCT, or UNION.

To satisfy the JDBC interface, executeQuery used on a non-query statement returns an empty ResultSet object with no columns.

Closing a Statement causes an open ResultSet object on that statement to be closed as well.

The cursor name for the cursor of a ResultSet can be set before the statement is executed. However, once it is executed, the cursor name cannot be altered.

java.sql.Statement Functionality Not Supported

Cloudscape does not implement the following JDBC 1.2 methods of java.sql.Statement:

- cancel
- getMaxFieldSize and setMaxFieldSize
- getMaxRows and setMaxRows
- setEscapeProcessing
- getQueryTimeout and setQueryTimeout

In addition, no SQL-J statements return multiple ResultSets; this means that getMoreResults always closes the current ResultSet and returns false.
Cloudbase provides all the required JDBC 1.2 type conversions and additionally allows use of the individual `setXXX` methods for each type as if a `setObject(Value, JDBCTypeCode)` invocation were made.

This means that `setString` can be used for any built-in target type.

The `setCursorName` method can be used on a `PreparedStatement` prior to an execute request to control the cursor name used when the cursor is created.

**Streaming Columns**

`setXXXStream` requests stream data between the application and the database.

JDBC allows an IN parameter to be set to a Java input stream for passing in large blobs of data in smaller chunks. When the statement is executed, the JDBC driver makes repeated calls to this input stream, reading its contents and transmitting those contents as the parameter data.

Cloudbase supports the three types of streams that JDBC 1.2 provides. These three streams are:

- `setBinaryStream` for streams containing uninterpreted bytes
- `setAsciiStream` for streams containing ASCII characters
- `setUnicodeStream` for streams containing Unicode characters

JDBC requires that you specify the length of the stream. Cloudbase does not have this restriction. Instead, you can provide the parameter `-1` to indicate “read until the end of the stream.” The stream object passed to these three methods can be either a standard Java stream object or the user’s own subclass that implements the standard `java.io.InputStream` interface.

Per the JDBC 1.2 standard, streams can be stored only in columns of the data types shown in Table 6-1. Streams cannot be stored in columns of the other built-in data types or of user-defined data types.

<table>
<thead>
<tr>
<th>Table 6-1 Streamable JDBC Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>AsciiStream</em></td>
</tr>
<tr>
<td>CHAR</td>
</tr>
<tr>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
A large X indicates the preferred target data type for the type of stream. (See Table 6-3, “Mapping of java.sql.Types to SQL-J Types,” on page 6-25.)

**NOTE:** In Version 3.0, if the stream is stored in a column of a type other than LONG VARCHAR or LONG VARBINARY, the entire stream must be able to fit into memory at one time. Streams stored in LONG VARCHAR and LONG VARBINARY columns do not have this limitation.

The following example shows how a user can store a streamed `java.io.File` in a LONG VARCHAR column:

```java
Statement s = conn.createStatement();
s.executeUpdate("CREATE TABLE atable (a INT, b LONG VARCHAR)");
conn.commit();
java.io.File file = new java.io.File("cloudscape.LOG");
int fileLength = (int) file.length();
// first, create an input stream
java.io.InputStream fin = new java.io.FileInputStream(file);
PreparedStatement ps = conn.prepareStatement(
    "INSERT INTO atable VALUES (?, ?)"");
ps.setInt(1, 1);
// set the value of the input parameter to the input stream
ps.setAsciiStream(2, fin, fileLength);
ps.execute();
conn.commit();
```
java.sql.CallableStatement

Cloudscape users do not typically use CallableStatements. In a Java program, it is more natural and easier to call methods with Statements or PreparedStatements and to retrieve the results, if any, through a ResultSet. For example, here’s an example of calling a static method that does not return a value:

```java
String archiveRecords = "CALL " +
    HotelStay.archiveRecords(CAST (? AS DATE))";
ps = conn.prepareStatement(archiveRecords);
ps.setDate(1, aDate.getDay(12));
ps.executeUpdate();
```

Here’s an example of calling a static method that returns a value (using a VALUES clause):

```java
Statement s = conn.createStatement();
ResultSet rs = s.executeQuery(
    " VALUES City.findCity(getCurrentConnection(), "+
    " CAST (? AS INT)");
rs.next();
City c = (city) rs.getObject(1);
```

However, Cloudscape allows you to use CallableStatements to execute methods and retrieve values, even though it is more natural to do so as demonstrated above.

**NEW:** Support for INOUT and OUT parameters through CallableStatement is new in Version 3.0.

Cloudscape supports all the JDBC 1.2 methods of CallableStatement:

- `getBoolean`
- `getByte`
- `getBytes`
- `getDate`
- `getDouble`
- `getFloat`
- `getInt`
- `getLong`
- `getObject`
- `getShort`
- `getString`
- `getTime`
CallableStatements and OUT Parameters

Cloudscape supports OUT parameters and CALL statements that return values, as in the following example:

```java
CallableStatement cs = conn.prepareCall(
    "? = CALL City.findCity(getCurrentConnection(), CAST (? AS INT))");
    cs.registerOutParameter(1, java.sql.Types.OTHER);
    cs.setInt(2, 35);
    cs.executeUpdate();
    City s = (City) cs.getObject(1);
```

**NOTE:** Using a CALL statement with a method that returns a value is only supported with the `? =` syntax.

Register the output type of the parameter before executing the call.

CallableStatements and INOUT Parameters

INOUT parameters are not a good fit for Java. Because of this, INOUT parameters map to an array of the parameter type in Java. (The method must take an array as its parameter.) This conforms to the recommendations of the SQLJ standard.

Given the following example:

```java
CallableStatement call = conn.prepareCall(
    "(CALL JBMSTours.Util).doubleIt(?))");
    // for inout parameters, it is good practice to
    // register the outparameter before setting the input value
    call.registerOutParameter(1, java.sql.Types.INTEGER);
    call.setInt(1,10);
    call.executeQuery();
    int retval = call.getInt(1);
```

the method `doubleIt` should take a one-dimensional array of ints. Here is sample source code for that method:

```java
public static void doubleMyInt(int[] i) {
    i[0] *=2;
    /* Cloudscape returns the first element of the array. */
```
NOTE: The return value is not wrapped in an array even though the parameter to the method is.

Table 6-2 INOUT Parameter Type Correspondence

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Array Type for Method Parameter</th>
<th>Value and Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>long[]</td>
<td>long</td>
</tr>
<tr>
<td>BINARY</td>
<td>byte[][]</td>
<td>byte[]</td>
</tr>
<tr>
<td>BIT</td>
<td>boolean[]</td>
<td>boolean</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date[]</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double[]</td>
<td>double</td>
</tr>
<tr>
<td>FLOAT</td>
<td>double[]</td>
<td>double</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int[]</td>
<td>int</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>byte[][]</td>
<td>byte[]</td>
</tr>
<tr>
<td>REAL</td>
<td>float[]</td>
<td>float</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short[]</td>
<td>short</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time[]</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp[]</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>TINYINT</td>
<td>byte[]</td>
<td>byte</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>byte[][]</td>
<td>byte[]</td>
</tr>
<tr>
<td>OTHER</td>
<td>yourType[]</td>
<td>yourType</td>
</tr>
</tbody>
</table>

Register the output type of the parameter before executing the call. For INOUT parameters, it is good practice to register the output parameter before setting its input value.
java.sql.ResultSet

A positioned update or delete issued against a cursor being accessed through a ResultSet object modifies or deletes the current row of the ResultSet object.

Some intermediate protocols, such as that in Cloudconnector, may pre-fetch rows. This causes positioned updates and deletes to operate against the row the underlying cursor is on, and not the current row of the ResultSet.

Cloudscape provides all the required JDBC 1.2 type conversions of the getXXX methods.

JDBC does not define the sort of rounding to use for ResultSet.getBigDecimal. Cloudscape uses java.math.BigDecimal.ROUND_HALF_DOWN.

Streaming Columns

If the underlying object is itself an OutputStream class, getBinaryStream returns the object directly.

To get a field from the ResultSet using streaming columns, you can use the getXXXStream methods if the type supports it. See Table 6-1 on page 6-15 for a list of types that support the various streams. (See also Table 6-3, “Mapping of java.sql.Types to SQL-J Types,” on page 6-25.)

You can retrieve data from one of the supported data type columns as a stream, whether or not it was stored as a stream.

The following example shows how a user can retrieve a LONG VARCHAR column as a stream:

```java
// retrieve data as a stream
ResultSet rs = s.executeQuery("SELECT b FROM atable");
while (rs.next()) {
    // use an InputStream to get the data
    InputStream ip = rs.getAsciiStream(1);
    // process the stream--this is just a generic way to
    // print the data
    int c;
    int columnSize = 0;
    byte[] buff = new byte[128];
    for (; ;) {
        int size = ip.read(buff);
        if (size == -1)
            break;
        // process the data
        c = buff[0];
        columnSize = 0;
        for (int i = 0; i < size; i++) {
            if (c == 127)
                break;
            columnSize = columnSize * 7 + c;
            c = buff[i+1];
        }
    }
    // print the data
    System.out.println("columnSize = " + columnSize);
    System.out.println("columnData = " + Arrays.toString(buff));
}
```
columnSize += size;
String chunk = new String(buff, 0, size);
System.out.print(chunk);
}
}
rs.close();
s.close();
conn.commit();
java.sql.ResultSetMetaData

Java Data Types

ResultSet metadata for columns that store Java data types and metadata items that describe the type of the column always return the following default values:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColumnDisplaySize</td>
<td>15</td>
</tr>
<tr>
<td>getPrecision</td>
<td>0</td>
</tr>
<tr>
<td>getScale</td>
<td>0</td>
</tr>
<tr>
<td>isAutoIncrement</td>
<td>false</td>
</tr>
<tr>
<td>isCaseSensitive</td>
<td>false</td>
</tr>
<tr>
<td>isCurrency</td>
<td>false</td>
</tr>
<tr>
<td>isSigned</td>
<td>false</td>
</tr>
</tbody>
</table>

For information on getting metadata about Java data types, see “Working with Metadata for Java Data Types” on page 9-1 in the Cloudscape Developer’s Guide.

Source and Updatability (all data types)

Cloudscape Version 3.0 does not track the source or updatability of columns in ResultSets, and so always returns the following constants for the following methods:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>isDefinitelyWritable</td>
<td>false</td>
</tr>
<tr>
<td>isReadOnly</td>
<td>false</td>
</tr>
<tr>
<td>isWritable</td>
<td>false</td>
</tr>
</tbody>
</table>
java.sql.SQLException

Cloudscape supplies values for the getMessage(), getSQLState(), and getErrorCode() calls of SQLExceptions. In addition, Cloudscape sometimes returns multiple SQLExceptions using the nextException chain. The first exception is always the most severe exception, with SQL-92 Standard exceptions preceding those that are specific to Cloudscape. For information on processing SQLExceptions, see “Working with Cloudscape SQLExceptions in an Application” on page 6-34 in the Cloudscape Developer’s Guide.
**java.sql.SQLWarning**

In Version 3.0, Cloudscape generates only one *SQLWarning*. Connection requests with the `create` attribute set to `true` get an *SQLWarning* if the database already exists.

All other informational messages are written to the Cloudscape system’s `cloudscape.LOG` file.
java.sql.Types

Table 6-3 shows the mapping of java.sql.Types to SQL-J types.

### Table 6-3 Mapping of java.sql.Types to SQL-J Types

<table>
<thead>
<tr>
<th>java.sql.Types</th>
<th>SQL-J Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>LONGINT</td>
</tr>
<tr>
<td>BINARY</td>
<td>BIT</td>
</tr>
<tr>
<td>BIT</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE PRECISION&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>LONG VARBINARY</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>LONG VARCHAR</td>
</tr>
<tr>
<td>NULL</td>
<td>Not a data type; always a value of a particular type</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>DECIMAL&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>OTHER</td>
<td>Java classes</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>BIT VARYING</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

- a. Values can be passed in using the FLOAT type code; however, these are stored as DOUBLE PRECISION values, and so always have the type code DOUBLE when retrieved.
- b. Values can be passed in using the NUMERIC type code; however, these are stored as DECIMAL values, and so always have the type code DOUBLE when retrieved.
JDBC 2.0-Only Features

JDBC 2.0 adds some functionality to the core API. This section documents the features supported by Cloudscape Version 3.0.

NEW: Beginning with Version 3.0, Cloudscape supports a subset of the JDBC 2.0 features for J2EE support. (For information about J2EE support, see “J2EE Overview” on page 8-1).

NOTE: These features are present only in a JDK1.2 environment.

These features are:


NOTE: The scroll insensitive functionality is implemented using an in-memory hash table. You may want to allocate additional memory for the JVM if using this feature with large ResultSets.


NOTE: Batch processing provides a performance benefit with some large client/server DBMSs. Batch updates do not provide any performance benefits in a Cloudscape system.

- “java.sql.Connection” on page 6-27
- “java.sql.ResultSet” on page 6-28
- “java.sql.Statement” on page 6-29
- “java.sql.PreparedStatement” on page 6-30
- “java.sql.CallableStatement” on page 6-31
- “java.sql.DatabaseMetaData” on page 6-32
- “java.sql.ResultSetMetaData” on page 6-33
- “java.sql.BatchUpdateException” on page 6-34
### Table 6-4 JDBC 2.0 Connection Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td><code>createStatement(int resultSetType, int resultSetConcurrency)</code></td>
<td>Returns <code>ResultSet.TYPE_FORWARD_ONLY</code> and <code>ResultSet.TYPE_SCROLL_INSENSITIVE</code> are the only scrolling types supported. If you request <code>TYPE_SCROLL_SENSITIVE</code>, Cloudscape issues an <code>SQLWarning</code> and returns a <code>TYPE_SCROLL_INSENSITIVE ResultSet</code>. <code>ResultSet.CONCUR_READ_ONLY</code> is the only concurrency supported. If you request a <code>CONCUR_UPDATABLE ResultSet</code>, Cloudscape issues an <code>SQLWarning</code> and returns a <code>CONCUR_READ_ONLY ResultSet</code>. (Use <code>ResultSet.getWarnings</code> to see warnings.)</td>
</tr>
<tr>
<td>PreparedStatement</td>
<td><code>prepareStatement(String sql, int resultSetType, int resultSetConcurrency)</code></td>
<td><code>ResultSet.TYPE_FORWARD_ONLY</code> is the only <code>resultSetType</code> supported. <code>ResultSet.CONCUR_READ_ONLY</code> is the only concurrency supported. If you request a <code>CONCUR_UPDATABLE ResultSet</code>, Cloudscape issues an <code>SQLWarning</code> and returns a <code>CONCUR_READ_ONLY ResultSet</code>.</td>
</tr>
<tr>
<td>CallableStatement</td>
<td><code>prepareCall(String sql, int resultSetType, int resultSetConcurrency)</code></td>
<td><code>ResultSet.TYPE_FORWARD ONLY</code> is the only <code>resultSetType</code> supported. <code>ResultSet.CONCUR read ONLY</code> is the only concurrency supported. If you request a <code>CONCUR_UPDATABLE ResultSet</code>, Cloudscape issues an <code>SQLWarning</code> and returns a <code>CONCUR_READ ONLY ResultSet</code></td>
</tr>
</tbody>
</table>
NOTE: When working with scrolling insensitive ResultSets when auto-commit mode is turned on, the only positioning method that can close the ResultSet automatically is the next() method. When auto-commit mode is on, his method automatically closes the ResultSet if it is called and there are no more rows. afterLast() does not close the ResultSet, for example.

### Table 6-5 JDBC 2.0 ResultSet Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>absolute(int row)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>afterLast()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>beforeFirst()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>beforeFirst()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>first()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getConcurrency()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getFetchDirection()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getFetchSize()</td>
<td>Always returns 1 in Version 3.0.</td>
</tr>
<tr>
<td>int</td>
<td>getRow()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isAfterLast()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isBeforeFirst</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isFirst()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isLast()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>last()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>previous()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>relative(int rows)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>setFetchDirection(int direction)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>setFetchSize(int rows)</td>
<td>A fetch size of 1 is the only size supported.</td>
</tr>
</tbody>
</table>
java.sql.Statement

Table 6-6  JDBC2.0  java.sql.Statement Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>addBatch(String sql)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>clearBatch()</td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>executeBatch()</td>
<td></td>
</tr>
</tbody>
</table>
### Table 6-7 JDBC 2.0 java.sql.PreparedStatement Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void</code></td>
<td><code>addBatch()</code></td>
<td></td>
</tr>
<tr>
<td><code>ResultSetMetaData</code></td>
<td><code>getMetaData()</code></td>
<td></td>
</tr>
</tbody>
</table>
### Table 6-8 JDBC 2.0 java.sql.CallableStatements Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigDecimal</td>
<td><code>getBigDecimal</code></td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td><code>getDate(int, Calendar)</code></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td><code>getTime(int, Calendar)</code></td>
<td></td>
</tr>
<tr>
<td>Timestamp</td>
<td><code>getTimeStamp(int, Calendar)</code></td>
<td></td>
</tr>
</tbody>
</table>
java.sql.DatabaseMetaData

Cloudscape implements all of the new JDBC 2.0 methods for this interface.
java.sql.ResultSetMetaData

Cloudscape implements all of the new JDBC 2.0 methods for this interface.
java.sql.BatchUpdateException

Thrown if there is a problem with a batch update.
JDBC Escape Syntax

JDBC provides a way of smoothing out some of the differences in the way different DBMS vendors implement SQL. This is called escape syntax. Escape syntax signals that the JDBC driver, which is provided by a particular vendor, scans for any escape syntax and converts it into the code that the particular database understands. This makes escape syntax DBMS-independent.

A JDBC escape clause begins and ends with curly braces. A keyword always follows the opening curly brace:

```
{keyword }
```

Cloudscape supports the following JDBC escape keywords, which are case-insensitive:

- “CALL” on page 6-36 (new in Version 3.0)
  The escape keyword for use in CallableStatements.
- “d” on page 6-37
  The escape keyword for date formats.
- “oj” on page 6-38
  The escape keyword for outer joins.
- “t” on page 6-39
  The escape keyword for time formats.
- “ts” on page 6-40
  The escape keyword for timestamp formats.

Other JDBC escape keywords are not supported.

**NOTE:** Cloudscape returns the SQL unchanged in the Connection.nativeSQL call, since the escape syntax is native to SQL-J. In addition, it is unnecessary to call Statement.setEscapeProcessing for this reason.
CALL
This syntax is supported for a `java.sql.Statement` and a `java.sql.PreparedStatement` in addition to a `CallableStatement`.

Syntax

```java
{ CALL statement }
```

Equivalent to

```java
CALL statement
```

Example

```java
{ CALL Util.doubleMyInt(?) }
```

```java
{ CALL ? = Util.returnDoubleMyInt(?) }
```
Cloudscape interprets the JDBC escape syntax for date as equivalent to the SQL-J syntax for dates.

**Syntax**
{d ‘yyyy-mm-dd’}

**Equivalent to**
DATE ‘yyyy-mm-dd’

**Example**
VALUES {d ‘1999-01-09’}
Cloudscape interprets the JDBC escape syntax for outer joins (and all join operations) as equivalent to the correct SQL-J syntax for outer joins or the appropriate join operation.

For information about join operations, see “JOIN operation” on page 1-89.

**Syntax**

{oj JOIN operation [JOIN operation ]* }

**Equivalent to**

JOIN operation [JOIN operation ]*

**Example**

-- outer join
SELECT *
FROM
{oj Countries LEFT OUTER JOIN Cities ON
(Countries.country_ISO_code=Cities.country_ISO_CODE)}

-- another join operation
SELECT *
FROM
{oj Countries JOIN Cities ON
(Countries.country_ISO_code=Cities.country_ISO_CODE)}

-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
SELECT country, city.getName(), hotel_name, normal_rate
FROM {oj Cities LEFT OUTER JOIN Countries
ON Countries.country_ISO_code=Cities.country_ISO_CODE
INNER JOIN Hotels
ON Cities.city_id=Hotels.city_id }
Cloudscape Version 3.0

Cloudscape interprets the JDBC escape syntax for time as equivalent to the correct SQL-J syntax for times.

Syntax
(t 'hh:mm:ss')

Equivalent to
TIME 'hh:mm:ss'

Example
VALUES {t '20:00:03'}
Cloudscape interprets the JDBC escape syntax for timestamp as equivalent to the correct SQL-J syntax for timestamps.

**Syntax**

\[
\text{ts 'yyyy-mm-dd hh:mm:ss.f...'}
\]

**Equivalent to**

\[
\text{TIMESTAMP 'yyyy-mm-dd hh:mm:ss.f...'}
\]

The fractional portion of timestamp constants (.f...) may be omitted.

**Example**

\[
\text{VALUES \{ts '1999-01-09 20:11:11.123455'}
\]
Cloudscape allows you to supply a list of attributes to its database connection URL, which is a JDBC feature. The attributes are specific to Cloudscape.

You typically set attributes in a semicolon-separated list following the protocol and subprotocol. For information on how you set attributes, see “The Cloudscape Database Connection URL Attributes” on page 6-9. This chapter provides reference information only.

The core attributes of the Cloudscape database connection URL are:

- “autocommit=false” on page 7-2
- “bootPassword=key” on page 7-3
- “create=true” on page 7-4
- “current=true” on page 7-5
- “databaseName=dbname” on page 7-6
- “dataEncryption=true” on page 7-7
- “logDevice=logDirectoryPath” on page 7-8
- “password=userPassword” on page 7-9
- “shutdown=true” on page 7-10
- “upgrade=true” on page 7-12
- “user=userName” on page 7-13
- “(no attributes)” on page 7-14

For information about additional attributes to the database connection URL in a synchronization system, see the Cloudscape Synchronization Guide.

**NOTE:** Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored.
Database Connection URL Attributes

**autocommit=false**

**Function**

Turns auto-commit off for the connection requested by the database connection URL. By default, auto-commit is set to true.

**Combining with Other Attributes**

You must specify a `databaseName` after the subprotocol in the URL or a `databaseName=nameofDatabase` attribute with this attribute.

**Example**

```
jdbc:cloudscape:toursDB;create=true;autocommit=false
```

```
jdbc:cloudscape:;databaseName=toursDB;autocommit=false
```
### bootPassword=key

**Function**

Specifies the key to use for encrypting a new database or booting an existing encrypted database. Specify an alphanumeric string at least eight characters long.

**Combining with Other Attributes**

When creating a new database, must be combined with `create=true` and `dataEncryption=true`. When booting an existing encrypted database, no other attributes are necessary.

```bash
-- boot an encrypted database
jdbc:cloudscape:encryptedDB;bootPassword=clo760uds2caPe

-- create a new, encrypted database
jdbc:cloudscape:newDB;create=true;dataEncryption=true;
    bootPassword=clo760uds2caPe
```
Database Connection URL Attributes

create=true

Function

Creates the standard (non-synchronization) database specified within the database connection URL Cloudscape system and then connects to it. If the database cannot be created, the error appears in the error log and the connection attempt fails with an SQLException indicating that the database cannot be found.

If the database already exists, creates a connection to the existing database and an SQLWarning is issued.

JDBC does not remove the database on failure to connect at create time if failure occurs after the database call occurs. If a database connection URL used create=true and the connection fails to be created, check for the database directory. If it exists, remove it and its contents before the next attempt to create the database.

Combining with Other Attributes

You must specify a databaseName (after the subprotocol in the database connection URL) or a databaseName=nameofDatabase attribute with this attribute.

You can combine this attributes with other attributes.

NOTE: If you specify create=true and the database already exists, an SQLWarning is raised.

Example

jdbc:cloudscape:toursDB;create=true

jdbc:cloudscape;;databaseName=newDB;create=true;autocommit=true
Function

Allows a Java method stored in the database to use the connection of the SQL-J statement that called it to make database-side JDBC calls (see “Requirements for Database-Side JDBC Methods Using Nested Connections” on page 5-3 in the Cloudscape Developer’s Guide).

The database name need not be specified.

When this attribute is used, the database connection URL is equivalent to jdbc:default:connection.

Combining with Other Attributes

Normally, do not combine with any attributes other than shutdown=true. If any other attributes are specified, they are ignored (except for shutdown). If databaseName is specified, it must be the same as the one in the original connection.

Example

jdbc:cloudscape:;current=true

jdbc:cloudscape:;current=true;shutdown=true
databaseName=nameofDatabase

**Function**

Specifies a database name for a connection; it can be used instead of specifying the database name in after the subprotocol.

For example, these URL (and Properties object) combinations are equivalent:

- jdbc:cloudscape:toursDB
- jdbc:cloudscape:;databaseName=toursDB
- jdbc:cloudscape:
  (with a property databaseName and its value set to toursDB in the Properties object passed into a connection request)

If the database name is specified both in the URL (as a subname) and as an attribute, the database name set as the subname has priority. For example, the following database connection URL connects to toursDB:

`jdbc:cloudscape:toursDB;databaseName=flightsDB`

Allowing the database name to be set as an attribute allows the `getPropertyInfo` method to return a list of choices for the database name based on the set of databases known to Cloudscape. For more information, see “java.sql.Driver.getPropertyInfo” on page 6-10.

**Combining with Other Attributes**

You can combine this attribute with all other attributes.

**Example**

`jdbc:cloudscape:;databaseName=newDB`
dataEncryption=true

**Function**

Specifies data encryption on disk for a new database.

**Combining with Other Attributes**

Must be combined with `create=true` and `bootPassword=key`.

**Example**

```
jdbc:cloudscape:encryptedDB;create=true;dataEncryption=true;
    bootPassword=cl0760uds2caPe
```
**Database Connection URL Attributes**

**logDevice=logDirectoryPath**

**Function**
During database creation, specifies path to the directory on which to store the database log. Even if specified as a relative path, the logDirectoryPath is stored internally as an absolute path.

**Combining with Other Attributes**
Use in conjunction with create=true or one of the synchronization-specific database creation attributes.

**Example**

```
jdbc:cloudscape:newDB;create=true;logDevice=d:/newDBlog
```

**NEW**: The logDevice attribute is new in Version 3.0
**password=userPassword**

**Function**
A valid password for the given user name.

**Combining with Other Attributes**
Use in conjunction with `user=userName`.

**Example**

```
jdbc:cloudscape:toursDB;user=jack;password=upTheHill
```
Database Connection URL Attributes

**shutdown=true**

**Function**

Shuts down the specified database if you specify a `databaseName`. (Reconnecting to the database reboots the database.)

Shuts down the database of the current connection if you use it in conjunction with the `current=true` attribute (the entire call to `DriverManager` must be executed within an SQL-J statement).

Shuts down the entire Cloudscape system if and only if you do not specify a `databaseName` or `current=true`.

When you are shutting down a single database, it lets Cloudscape perform a final checkpoint on the database.

When you are shutting down a system, it lets Cloudscape perform a final checkpoint on all system databases, deregister the JDBC driver, and shut down within the JVM before the JVM exits. A successful shutdown always results in an `SQLException` indicating that Cloudscape has shut down and that there is no connection. Once Cloudscape is shut down, you can restart it by reloading the driver. For details on restarting Cloudscape, see “Shutting Down the System” on page 2-8 in the *Cloudscape Developer’s Guide*.

**Checkpointing** means writing all data and transaction information to disk so that no recovery needs to be performed at the next connection.

Used to shut down the entire system only when it is embedded in an application. Shut down a server framework the prescribed way; see the *Cloudscape Server and Administration Guide*.

**NOTE:** Any request to the `DriverManager` with a `shutdown=true` attribute raises an exception.

You can issue the `DriverManager.getConnection` request with a `shutdown=true` attribute within a connection, and thus within an SQL-J statement:

```java
-- shuts down current database
CALL (CLASS java.sql.DriverManager).getConnection(
    'jdbc:cloudscape:;current=true;shutdown=true');
```

**Combining with Other Attributes**

Combine with `current=true` to shut down the current database.
When not combined with other attributes or \textit{databaseName}, it shuts down the Cloudscape system.

\textbf{Example}

\begin{itemize}
  \item -- shuts down entire system
    \texttt{jdbc:cloudscape;;shutdown=true}
  \item -- shuts down salesDB
    \texttt{jdbc:cloudscape:;salesDB;shutdown=true}
  \item -- shuts down current database
    \texttt{jdbc:cloudscape;;current=true;shutdown=true}
\end{itemize}
**Database Connection URL Attributes**

**upgrade=true**

Allows upgrade of a database created with a version of Cloudscape prior to 3.0. If you try to connect to a database created with a version of Cloudscape prior to 3.0, an exception is thrown and the connection is disallowed unless you specify this attribute.

If no upgrade is needed, the attribute is ignored.

Once a database has been upgraded to a 3.0 database, it is unusable with prior versions of Cloudscape.

**Example**

```
jdbc:cloudscape:toursDB;upgrade=true
```
user=user={Name}

Specifies a valid user name for the system, specified with a password. A valid user name and password are required when user authentication is turned on.

Combining with Other Attributes

Use in conjunction with password=userPassword.

Example

The following database connection URL connects the user jill to toursDB:

```
jdbc:cloudscape:toursDB;user=jill;password=toFetchAPail
```
**Database Connection URL Attributes**

*(no attributes)*

If no attributes are specified, you must specify a `databaseName`.

Cloudscape opens a connection to an existing database with that name in the current system directory. If the database does not exist, the connection attempt returns an `SQLException` indicating that the database cannot be found.

**Examples**

```
jdbc:cloudscape:mydb
```
J2EE Overview

J2EE, or the Java 2 Platform, Enterprise Edition, is a new standard for development of enterprise applications; one example is Enterprise Java Beans (EJBs) with distributed capabilities.

Cloudscape Version 3.0 is a J2EE-conformant component in a distributed J2EE system. As such, it is one part of a larger system that includes, among other things, a JNDI server, a connection pool module, a transaction manager, a resource manager, and user applications. (Cloudscape also supports the JTA API, which is not a current J2EE requirement, but this functionality provides another piece of the same system.) Within this system, Cloudscape can serve as the resource manager.

For more information on J2EE, see the J2EE specification available at java.sun.com/j2ee/docs.html.

In order to qualify as a resource manager in a J2EE system, J2EE requires these basic areas of support:
- **JNDI support.**
  Allows calling applications to register names for databases and access them through those names instead of through database connection URLs. Implementation of one of the JDBC extensions, `javax.sql.DataSource`, provides this support.

- **Connection pooling**
  A mechanism by which a connection pool server keeps a set of open connections to a resource manager (Cloudscape). A user requesting a connection can get one of the available connection from the pool. Such a connection pool is useful in client/server environments because establishing a connection is relatively expensive. In an embedded environment, connections are much cheaper. Implementation of two of the JDBC extensions, `javax.sql.ConnectionPoolDataSource` and `javax.sql.PooledConnection`, provide this support.

- **XA support.**
  XA is one of several standards for distributed transaction management. It is based on two-phase commit. The `javax.sql.XAxxx` interfaces, along with `java.transaction.xa` package, are an abstract implementation of XA. For more information about XA, see *X/Open CAE Specification—Distributed Transaction Processing: The XA Specification*, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3. Implementation of the JTA API, the interfaces of the `java.transaction.xa` package (`javax.sql.XAConnection`, `javax.sql.XADataSource`, `javax.transaction.xa.XAResource`, `javax.transaction.xa.Xid`, and `javax.transaction.xa.XAException`), provide this support.

- **A subset of the 2.0-only additions to the core JDBC API.** (See “JDBC 2.0-Only Features” on page 6-26.)

With the exception of the core JDBC interfaces, these interfaces are not visible to the end-user application; instead, they are used only by the other back-end components in the system.

**NOTE:** For information on the classes that implement these interfaces and how to use Cloudscape as a resource manager, see Chapter 7, “Using Cloudscape as a J2EE Resource Manager” in the *Cloudscape Developer’s Guide*.

## Requirements

These features require the following:
The JTA API

The JTA API is made up of the two interfaces and one exception that are part of the java.transaction.xa package. Cloudscape fully implements this API.

- javax.transaction.xa.XAResource
- javax.transaction.xa.Xid
- javax.transaction.xa.XAException

NEW: JTA support is new in Version 3.0.

Notes on Product Behavior

Recovered Global Transactions

Using the XAResource.prepare call causes a global transaction to enter a prepared state, which allows it to be persistent. Typically, the prepared state is just a transitional state before the transaction outcome is determined. However, if the system crashes, recovery puts transactions in the prepared state back into that state and awaits instructions from the transaction manager.

XAConnections and User Names and Passwords

If a user opens an XAConnection with a user name and password, the transaction it created cannot be attached to an XAConnection opened with a different user name and password. A transaction created with an XAConnection without a user name and password can be attached to any XAConnection.
However, the user name and password for recovered global transactions are lost; any XAConnection can commit or rollback that in-doubt transaction.

**NOTE:** Cloudscape’s XADataSources can only be local. No remote (client/server) support is provided.
javax.sql: JDBC Extensions

This section documents the JDBC extensions that Cloudscape implements for J2EE compliance.

NEW: Support for these JDBC extensions is new in Version 3.0.

- `javax.sql.DataSource`
  Cloudscape’s implementation of `DataSource` means that it supports JNDI; as a resource manager, it allows a database to be named and registered within a JNDI server. This allows the calling application to access the database by a name (as a data source) instead of through a database connection URL.

NOTE: Support for remote (client/server) data sources is provided through RmiJdbc. (XADataSources are local only.)

- `javax.sql.ConnectionPoolDataSource`
  Support of this interface allows a connection pool server to maintain a set of connections to the resource manager (Cloudscape). Such a connection pool is useful in client/server environments because establishing a connection is relatively expensive. In an embedded environment, connections are much cheaper.

- `javax.sql.PooledConnection`
  Support of this interface allows a connection pool server to maintain a set of connections to the resource manager (Cloudscape). Such a connection pool is useful in client/server environments because establishing a connection is relatively expensive. In an embedded environment, connections are much cheaper.

- `javax.sql.XAConnection`
  An `XAConnection` produces an `XAResource`, and, over its lifetime, many `Connections`. It allows for distributed transactions.

- `javax.sql.XADataSource`
  An `XADataSource` is simply a `ConnectionPoolDataSource` that produces `XACredentials`.

In addition, Cloudscape provides three methods for `XADataSource`, `DataSource`, and `ConnectionPoolDataSource` that are not part of the standard API:

- `setCreateDatabase(String create)`
  Sets a property to create a database at the next connection. The string argument must be “create”.
setShutdownDatabase(String shutdown)
Sets a property to shut down a database. Shuts down the database at the next connection. The string argument must be “shutdown”.

setRemoteDataSourceProtocol(String rmi)
Sets a property to enable the correct protocol for working with data sources in a client/server environment using RmiJdbc. The string must be “rmi”. Not valid for XADatasources.

**NOTE:** Set these properties before getting the connection.
Appendix A  Cloudscape API

Cloudscape provides Javadoc HTML files of API classes and interfaces in the javadoc subdirectory. This appendix provides a brief overview of the API. Cloudscape does not provide the Javadoc for the java.sql packages, the main API for working with Cloudscape, because that is part of the JDK. For information about Cloudscape’s implementation of JDBC, see Chapter 6, “JDBC Reference”.

This document divides the API classes and interfaces into three categories:

- **“Stand-Alone Tools and Utilities” on page A-1**
  These are Java classes that stand on their own and are invoked in a command window.

- **“Core Classes” on page A-2**
  You do not invoke these on the command-line, only within a specified context within another application. These classes are boot classes or classes that implement javax.sql.DataSource and related APIs.

- **“SQL-J Extensions” on page A-3**
  These are interfaces that are implemented internally by Cloudscape. You use these interfaces only within SQL-J statements.

### Stand-Alone Tools and Utilities

These classes are in the packages COM.cloudscape.tools and COM.cloudscape.util.

- **COM.cloudscape.tools.ij**
  An SQL scripting tool that can run as an embedded or a remote client/server application. See the Cloudscape Tools and Utilities Guide.
Cloudscape API

- **COM.cloudscape.tools.sysinfo**
  A command-line, server-side utility that displays information about your JVM and Cloudscape product. See the Cloudscape Tools and Utilities Guide.

- **COM.cloudscape.tools.cview**
  A user interface for creating and managing Cloudscape databases can run as an embedded or a remote client/server application. See the Cloudscape Tools and Utilities Guide.

- **COM.cloudscape.tools.dbclasses**
  A command-line, server-side utility that allows you to store application logic in a database. See the Cloudscape Tools and Utilities Guide.

- **COM.cloudscape.util.DBClassLoad**
  DBClassLoad enables database class loading for applications using Cloudscape in embedded mode. See the Cloudscape Tools and Utilities Guide.

- **COM.cloudscape.tools.FileExport**
  A server-side utility for exporting data. See the Cloudscape Tools and Utilities Guide.

- **COM.cloudscape.tools.FileImport**
  A server-side utility for importing data. See the Cloudscape Tools and Utilities Guide.

- **COM.cloudscape.tools.databasePreloader**
  A client-side utility for pre-loading Cloudview’s stored prepared statements into a database. Makes the first use of the database with Cloudview faster.

Core Classes

Boot Classes

These classes are part of the COM.cloudscape.core package:

- **COM.cloudscape.core.JDBCDriver**
  Used to boot the local, built-in JDBC driver and the Cloudscape system. See the Cloudscape Developer’s Guide.
• **COM.cloudscape.core.WebLogicDriver**
  Used to boot the remote JDBC driver for a Cloudconnector client. See the *Cloudscape Server and Administration Guide*.

• **COM.cloudscape.core.RmiJdbcDriver**
  Used to boot the remote JDBC driver for an RmiJdbc client. See the *Cloudscape Server and Administration Guide*.

• **COM.cloudscape.core.ChannelSystem**
  Used to boot Cloudscape running in a Marimba channel.

• **COM.cloudscape.core.CloudscapeServer**
  Used to boot Cloudconnector. See the *Cloudscape Server and Administration Guide*.

• **COM.cloudscape.core.ServletHandler**
  The optional refresh service for source databases running inside of Cloudconnector. For more information, see the *Cloudscape Synchronization Guide*.

**Data Source Classes**

These classes are all related to Cloudscape’s implementation of *javax.sql.DataSource* and related APIs. For more information, see Chapter 7, “Using Cloudscape as a J2EE Resource Manager” *Cloudscape Developer’s Guide*. These classes are part of the **COM.cloudscape.core** package:

• **COM.cloudscape.core.DataSourceFactory**

• **COM.cloudscape.core.AbstractDataSource**

• **COM.cloudscape.core.BasicDataSource**

• **COM.cloudscape.core.LocalConnectionPoolDataSource**

• **COM.cloudscape.core.XaDataSource**

**SQL-J Extensions**

Classes and interfaces in this category are used only within SQL-J statements and thus can be used in an embedded or a client/server system.
Types Used in System Tables

These are the Java data types supplied by Cloudscape and stored in system tables, or used or generated by types stored in system tables. An asterisk indicates a type used by but not stored in the system table types.

- `COM.cloudscape.types.AliasInfo` Used in `SYS.SYSALIASES`. Provides information about an alias (method alias, class alias, work unit, or user-defined aggregate).
- `COM.cloudscape.types.DependableFinder` Used in `SYS.SYSDEPENDS`. Provides information about database object dependencies.
- `COM.cloudscape.types.TypeDescriptor` Used in `SYS.SYSCOLUMNS`. Provides precision, length, scale, nullability, type name, and storage type of data. Also used by user-defined aggregates.
- `COM.cloudscape.types.IndexDescriptor` Used in `SYS.SYSCONGLOMERATES`. Describes indexes.
- `COM.cloudscape.types.ReferencedColumnsDescriptor` Used in `SYS.SYSTRIGGERS`. Describes columns referenced by triggers.
- `COM.cloudscape.types.DatabaseInstant` Used in `SYS.SYSSYNCINSTANTS`. Identifies a moment in time for a database.
- `COM.cloudscape.types.Dependable*`
- `COM.cloudscape.types.UUID*`
- `COM.cloudscape.system.UUIDFactory*`

Other Types

- `COM.cloudscape.database.PropertyInfo` Used to set database-wide and conglomerate-specific properties. See Tuning Cloudscape.
- `COM.cloudscape.database.UserUtility` Used to maintain authorization lists. See the Cloudscape Developer’s Guide.
• \texttt{COM.cloudscape.types.RuntimeStatistics}
  Used to display information about the execution of a statement. See \textit{Tuning Cloudscape}.

• \texttt{COM.cloudscape.util.BitUtil}
  A useful utility for working with bits. See “Use the Cloudscape-Supplied Methods to Work with Bits” on page 10-5.

• \texttt{COM.cloudscape.database.Database}
  Provides some utility methods for database-level actions, such as backing up a database.

• \texttt{COM.cloudscape.database.Factory}
  A class that returns instances of \texttt{Databases} or \texttt{Systems} so that methods of \texttt{COM.cloudscape.database.Database} or \texttt{COM.cloudscape.system.System} may be called.

• \texttt{COM.cloudscape.system.System}
  Provides some utility methods for system-level actions.

• \texttt{COM.cloudscape.database.TriggerExecutionContext}
  Provides information about a trigger’s context.

• \texttt{COM.cloudscape.database.ConsistencyChecker}
  Provides utility methods for consistency checking.

\textbf{SQLException-Related}

• \texttt{COM.cloudscape.tools.ImportExportSQLException}

• \texttt{COM.cloudscape.database.JavaSQLException}

• \texttt{COM.cloudscape.database.JBMSException}

• \texttt{COM.cloudscape.types.JBMSExceptionSeverity}

\textbf{User-Throwable}

• \texttt{COM.cloudscape.authentication.Interface.AuthenticationException}

• \texttt{COM.cloudscape.synchronization.StopRefreshSQLException}

• \texttt{COM.cloudscape.synchronization.SkipTransactionSQLException}
Interfaces for User-Defined Aggregates

These types allow you to make a Java data type usable within aggregate statements such as min, max, sum, and so on.

- `COM.cloudscape.aggregates.AggregateDefinition`
- `COM.cloudscape.aggregates.Aggregator`

VTIs

Virtual Table Interfaces, also known as VTIs, are classes that implement `java.sql.ResultSet` and that allow you to access data external to Cloudscape as if it were stored in the database. For more information, see “Getting External Data: Using the Cloudscape Virtual Table Interface” on page 4-19 in the Cloudscape Developer’s Guide.

Cloudscape also provides some built-in VTIs that make it easy to work with some internal system information.

- `COM.cloudscape.vti.LockTable`
  A built-in VTI class that provides lock debugging information.
- `COM.cloudscape.vti.TransactionTable`
  A built-in VTI class that provides lock debugging information.
- `COM.cloudscape.vti.TriggerOldTransitionRows`
  A built-in VTI class that provides information for triggers.
- `COM.cloudscape.vti.TriggerNewTransitionRows`
  A built-in VTI class that provides information for triggers.
- `COM.cloudscape.synchronization.RowListVTI`
- `COM.cloudscape.synchronization.StatementListVTI`
- `COM.cloudscape.synchronization.TransactionListVTI`
  Utility for extracting information from `SYS.SYSERRORS`.
- `COM.cloudscape.vti.VTICosting`
  If a VTI class implements this interface, Cloudscape’s optimizer can use the costing information it provides.
- `COM.cloudscape.vti.VTIEnvironment`
  Not used in this release.
- `COM.cloudscape.vti.ExternalQuery`
  An example VTI class that accesses data from any JDBC source.
Miscellaneous Utilities and Interfaces

- `COM.cloudscape.vti.VTITemplate`
  A useful class for building a VTI class (JDBC 1.2).
- `COM.cloudscape.vti.VTIMetaDataTemplate`
  A useful class for building a VTI class (JDBC 1.2).
- `COM.cloudscape.vti20.VTITemplate`
  A useful class for building a VTI class (JDBC 2.0).
- `COM.cloudscape.vti20.VTIMetaDataTemplate`
  A useful class for building a VTI class (JDBC 2.0).

Miscellaneous Utilities and Interfaces

- `COM.cloudscape.util.DriverUtil`
  A JDBC application utility that understands Cloudscape database connection URLs and drivers.
- `COM.cloudscape.util.JDBCDisplayUtil`
  A JDBC application utility for formatting JDBC ResultSets.
- `COM.cloudscape.authentication.Interface.AuthenticationScheme`
  An interface provided by Cloudscape. Classes that provide an alternate user authentication scheme must implement this interface. For information about users, see “Working with User Authentication” on page 8-5 in the Cloudscape Developer’s Guide.
- `COM.cloudscape.authentication.util.SimpleAuthenticationScheme`
  A sample class that provides an alternative user authentication scheme and that implements `COM.cloudscape.authentication.Interface.AuthenticationScheme`.
- `COM.cloudscape.util.KeyGen`
  A utility for generating unique keys in a distributed system. For more information, see the Cloudscape Synchronization Guide.
Symbols
% as wildcard in SQL-J 1-2
* as wildcard in SQL-J SELECT 1-2, 1-103
+,-,.*,./, unary + and - expressions 1-219
-- (comment delimiters within SQL-J) 1-2
? (dynamic parameter)
   in prepared statements 1-226
   used in conditional expression 1-228
?: (conditional) expression 1-145
_ as wildcard within SQL-J 1-2
{oj (JDBC escape syntax) 6-38

A
AggregateName identifier 1-10
Aggregates 1-129
   and data types 1-129
   definition 1-129
   requirements for 1-129
AliasInfo class alias 1-40
ALTER STATEMENT statement 1-18
ALTER TABLE statement 1-20
AND boolean operator 1-223
ASC
   implied in ORDER BY clause 1-93
AT ISOLATION clause 1-24
AuthorizationIdentifierName identifier 1-11
autocommit=false (database connection URL attribute) 7-2
AVG aggregate function 1-131

B
Backing indexes
   created automatically to enforce primary key, unique, and foreign key constraints
   1-33, 1-41
   specifying storage properties for 1-33
Batch processing (JDBC) 6-29, 6-30, 6-31
BETWEEN boolean operator 1-224
Bit strings
   variable-length 1-192
BIT VARYING data type 1-192
BIT_LENGTH function 1-133, 1-219
BitUtil class alias 1-40
BOOLEAN data type 1-193
Boolean expression 1-223
Boolean operators
   part of WHERE clause 1-223
Booleans
   assigning and comparing to numeric types 1-188
   bootPassword=true (database connection URL attribute) 7-3

C
CALL (JDBC escape keyword) 6-36
CALL statement 1-25–1-28
   and return values 1-25
   used to invoke work units 1-26
Case sensitivity of keywords and identifiers 1-2
CAST expression 1-218
CAST function 1-134–1-142
CHAR data type 1-194
CHAR_LENGTH function 1-143, 1-219
Character expressions 1-220
CHARACTER_LENGTH function 1-143, 1-219
Check constraints 1-33
   use of IN expression within 1-34
Class aliases
   ambiguity with column names 1-235
   built-in 1-40
   creating 1-39
   default 1-39
   dropping 1-65
Class path
accessing read-only databases from 6-6
ClassAlias identifier 1-9
Column constraints
used in CREATE TABLE statement 1-29
Column defaults 1-49
Column reference expression 1-217
Column-level constraints 1-29
ColumnName identifier 1-7
Columns
adding 1-20
COM.cloudscape.authentication.Interface.AuthenticationScenario A-7
COM.cloudscape.core.AbstractDataSource A-3
COM.cloudscape.core.BasicDataSource A-3
COM.cloudscape.core.ChannelSystem A-3
COM.cloudscape.core.DataSourceFactory A-3
COM.cloudscape.core.JDBCDriver A-2
COM.cloudscape.core.LocalConnectionPoolDataSource A-3
COM.cloudscape.core.ServletHandler A-3
COM.cloudscape.core.WebLogicDriver A-3
COM.cloudscape.core.XaDataSource A-3
COM.cloudscape.database.AggregateDefinition A-6
COM.cloudscape.database.Aggregator A-6
COM.cloudscape.database.Factory A-5
COM.cloudscape.database.JavaSQLException A-5
COM.cloudscape.database.JBMSException A-5
COM.cloudscape.database.PropertyInfo A-4
COM.cloudscape.database.Utility A-4
COM.cloudscape.synchronization.RowListVTI A-6
COM.cloudscape.synchronization.SkipTransactionSQLException A-5
COM.cloudscape.synchronization.StatementListVTI A-6
COM.cloudscape.synchronization.StopRefreshSQLException A-5
COM.cloudscape.synchronization.TransactionListVTI A-6
COM.cloudscape.system.System A-5
COM.cloudscape.system.UUIDFactory A-4
COM.cloudscape.tools.CloudscapeServer A-3
COM.cloudscape.tools.cview A-2
COM.cloudscape.tools.databasePreloader A-2
COM.cloudscape.tools.dbclasses A-2
COM.cloudscape.tools.DBClassLoad A-2
COM.cloudscape.tools.FileExport A-2
COM.cloudscape.tools.FileImport A-2
COM.cloudscape.tools.ImportExportSQLException A-5
COM.cloudscape.tools.sysinfo A-2
COM.cloudscape.TriggerExecutionContext A-5
COM.cloudscape.types.DatabaseInstant A-4
COM.cloudscape.types.Dependable A-4
COM.cloudscape.types.DependableFinder A-4
COM.cloudscape.types.ErrorInfo A-4
COM.cloudscape.types.IndexDescriptor A-4
COM.cloudscape.types.JBMSExceptionSeverity A-5
COM.cloudscape.types.ReferencedColumnDescriptor A-4
COM.cloudscape.types.RuntimeStatistics A-5
COM.cloudscape.types.RuntimeStatisticsInterface A-161
COM.cloudscape.types.SyncErrorInfo A-4
COM.cloudscape.types.TypeDescriptor A-4
COM.cloudscape.types.UUID A-4
COM.cloudscape.util.BitUtil A-5
COM.cloudscape.util.DriverUtil A-7
COM.cloudscape.util.JDBCDisplayUtil A-7
COM.cloudscape.util.KeyGen A-7
COM.cloudscape.vti.ExternalQuery A-6
COM.cloudscape.vti.LockTable A-6
COM.cloudscape.vti.TransactionTable A-6
COM.cloudscape.vti.TriggerNewTransitionRows A-6
COM.cloudscape.vti.TriggerOldTransitionRows A-6
COM.cloudscape.vti.VTICosting A-6
COM.cloudscape.vti.VTIEncoding A-6
COM.cloudscape.vti.VTITemplate A-7
Comment delimiters within SQL-J 1-2
Compilation stages of 1-163
Concatenation function 1-144
as expression 1-220
Conditional expression 1-145, 1-218
Connecting to a database (using databaseName attribute) 7-6
Connection pooling 8-2, 8-5

Cloudscape Reference Manual
Connections
  garbage collection of 6-11
  nested 1-242
  session-severity exceptions closing 6-11
  system-severity exceptions closing 6-11
ConsistencyChecker class alias 1-40
CONTRAINT clause 1-29
ConstraintName identifier 1-8
Constraints 1-29-1-37
  adding 1-20
  column-level 1-29
  differences between column-level and table-level 1-30
  disabling 1-35, 1-109
  dropping 1-20
  enabling 1-109
  getting information about key columns 4-10
  naming 1-31
  recommendation to name 1-31
  table-level 1-29
Conversions
  implicit (to and from strings) 1-186
  of Java data types using CAST 1-140
  of SQL-92 data types using CAST 1-134
  using CAST function (explicit) 1-134
CorrelationName identifier 1-6
COUNT aggregate function 1-146
COUNT(asterisk) aggregate function 1-147
CREATE AGGREGATE statement 1-38
CREATE CLASS ALIAS statement 1-39
CREATE INDEX statement 1-41-1-42
CREATE METHOD ALIAS statement 1-43
CREATE SCHEMA statement 1-44
CREATE STATEMENT statement 1-45
CREATE TABLE statement 1-48-1-50
CREATE TRIGGER statement 1-51
CREATE VIEW statement 1-60
create=true (database connection URL attribute) 7-4
CROSS JOIN expression 1-62
current=true (database connection URL attribute) 7-5
CURRENT_DATE function 1-148
  as expression 1-220
CURRENT_TIME function 1-149
  as expression 1-220
CURRENT_TIMESTAMP function 1-150
  expression 1-220
CURRENT_USER (function) 1-151
CursorName identifier 1-8
Cursors 1-78
  pre-fetching affecting 6-20
  scrolling insensitive 6-27
D
  d (JDBC escape keyword) 6-37
Data sources 8-5
  creating a database for 8-5
  setting the correct protocol for RmiJdbc client/server 8-6
  shutting down a database for 8-6
Database connection URL attributes 6-9
  for Cloudconnector clients 6-5
  settable through Properties in connection call 6-9
  syntax in client/server environments 6-6, 6-7
  syntax in embedded mode 6-5
databaseName=nameOfDatabase (database connection URL attribute) 7-6
Databases
  encrypting 7-3, 7-7
dataEncryption=true (database connection URL attribute) 7-7
  DATE data type 1-195
  Date/time expressions 1-220
  dbclasses class alias 1-40
  DECIMAL data type 1-196
  DefaultInfo class alias 1-40
  Defaults 1-49
    adding 1-20
    adding or dropping 1-20, 1-22
    dropping 1-20
  DELETE statement 1-63
  Delimited identifiers
    definition 1-2
  DependableFinder class alias 1-40
Dependencies
  internal tracking of 1-15
  not tracked for Java data types 1-17
DESC
  optional element of ORDER BY clause 1-93
Dictionary objects
  definition 1-3
DISTINCT
  qualifier in AVG function 1-131
  qualifier in COUNT aggregate function 1-146
DISTINCT clause
  part of SelectExpression 1-102
Distributed transaction management 8-2
DOUBLE PRECISION data type 1-198
Driver
  for Cloudconnector clients 6-4
for RmiJdbc clients 6-4
DROP AGGREGATE statement 1-64
DROP CLASS ALIAS statement 1-65
DROP INDEX statement 1-65, 1-66
DROP METHOD ALIAS statement 1-67
DROP SCHEMA statement 1-68
DROP STATEMENT statement 1-69
DROP TABLE statement 1-70
DROP TRIGGER statement 1-71
DROP VIEW statement 1-72
Dynamic parameter expression 1-218
Dynamic parameters 1-226
need to cast data type of within method invocation 1-234

E
Encrypting databases 7-3, 7-7
Entity-relationship diagram for system tables 4-2
ErrorInfo class alias 1-40
ErrorTransactionVTI class alias 1-40
Escape syntax (JDBC) 6-35
EXECUTE STATEMENT statement 1-73
EXISTS boolean operator 1-224
Expression subquery 1-101
Expressions
hierarchy of 1-221
in SQL-J language 1-217
precedence of 1-221
External virtual tables 1-76–1-77
ExternalQuery class alias 1-40
EXTRACT function 1-152
as expression 1-220

F
Factory class alias 1-40
Field access 1-238–1-239
Field access expression 1-218
FileExport class alias 1-40
FileImport class alias 1-40
FLOAT data type
as alias for REAL or DOUBLE PRECISION 1-199
java.sql type converted to DOUBLE PRECISION when retrieved 6-25
FOR UPDATE clause 1-78
Foreign keys 1-32
FROM clause 1-79
Functions, built-in 1-128–1-174

G
getAsciiStream 6-20
getBinaryStream 6-20
GETCURRENTCONNECTION() function 1-242
getUnicodeStream 6-20
getXXXStream requests 6-20
GROUP BY clause 1-80

H
HAVING clause 1-82

I
Identifiers
definition 1-3
IN (?) or dynamic) parameters 1-226
IN boolean operator 1-224
Index class alias 1-40
IndexDescriptor class alias 1-40
Indexes
and Java data types 1-42
automatic creation of for primary key, unique, and foreign key constraints 1-33, 1-41
getting information about key columns 4-10
specifying page size for 1-41
IndexName identifier 1-8
INNER JOIN expression 1-83
In-place updates 1-78
INSERT statement 1-86
INSTANCEOF boolean operator 1-225
INSTANCEOF expressions 1-240
INTEGER data type 1-200
Integer literals
using parentheses to avoid interpretation as DOUBLE PRECISION 1-235
IS boolean operator 1-225
IS NOT NULL boolean operator 1-223
IS NULL boolean operator 1-223
Isolation levels
changing for a single statement 1-24
changing for a transaction 1-114
setting via JDBC 6-11

J
J2EE 8-1–8-6
JarName identifier 1-11
Java data types 1-211–1-213
and equality comparisons 1-213
assignability of 1-212
built-in 1-212
metadata for 6-22
ordering by 1-214
requirements for ordering 1-214
used in method invocation 1-231

Cloudscape Reference Manual
Java expressions 1-221, 1-230–1-242
Java identifiers
  case sensitivity of within SQL-J 1-2
java.sql.CallableStatement 6-17
java.sql.CallableStatement
  and INOUT parameters 6-18
  and OUT parameters 6-18
java.sql.Connection 6-11
java.sql.Connection.isReadOnly 6-11
java.sql.Connection.setReadOnly 6-11
java.sql.Connection.setTransactionIsolation 6-11
java.sql.DatabaseMetaData 6-12
java.sql.Driver 6-3
java.sql.DriverManager.getConnection 6-5
java.sql.PreparedStatement 6-15
java.sql.ResultSet 6-20
java.sql.ResultSetMetaData 6-22
java.sql.SQLException 6-23
java.sql.SQLWarning 6-24
java.sql.Statement 6-24
java.sql.Types
  mapped to SQL-J types 6-25
java.transaction.xa.Connection 8-2
JavaClassName identifier 1-12
JavaFieldIdentifier identifier 1-13
JavaIdentifiers
  definition 1-3
  rules for 1-3–1-14
JavaMethodName identifier 1-13
javax.sql.ConnectionPoolDataSource 8-2
javax.sql.ConnectionPoolDataSource 8-5
javax.sql.ConnectionPoolDataSource 8-2, 8-5
javax.sql.ConnectionPoolDataSource 8-2, 8-5
javax.sql.PooledConnection 8-5
javax.sql.PooledConnection 8-2
javax.sql.XAConnection 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
javax.sql.XADataSource 8-5
java.transaction.xa.XAException 8-3
java.transaction.xa.XAException 8-3
java.transaction.xa.XAException 8-3
java.sql.SQL JTypes 6-25
jdbc.drivers system property
  using to load driver 6-3
JNDI 8-2
JOIN operation 1-89
JTA API 8-3

K
KeyGen class alias 1-40
Keywords 2-1–2-4
  case insensitivity of 1-2

L
LEFT OUTER JOIN expression 1-90
LIKE boolean operator 1-224
Literal expression 1-217
Lock escalation
  by user 1-91
Lock granularity
  specifying when you create a table 1-48
LOCK TABLE statement 1-91
LockTable class alias 1-40
logDevice=logDirectoryPath (database
  connection URL attribute) 7-8
LONG BINARY (data type) 1-201
LONG BIT VARYING (data type) 1-201
LONG VARCHAR VARYING (data type) 1-202
LONGVARBINARY 6-25
LONGVARCHAR 6-25
LOWER (function) 1-153
LTRIM (function) 1-154

M
Mapping
  of Java to SQL-J types 1-245
  of SQL-J data types to Java data types 1-244
  of SQL-J types to java.sql.Types 6-25
Matching index scans 1-165
MAX aggregate function 1-155
Method aliases 1-43
  dropping 1-67
Method invocation 1-231–1-235
  and dynamic parameters 1-234
  as expression 1-218
  runtime exceptions generating statement
    exceptions 1-234
  type conversion during 1-232
Method receiver
  definition 1-231

Cloudscape Version 3.0
Method resolution
and type correspondence 1-246
MethodAliasName identifier 1-9
Methods returning void types
executable only within CALL statements 1-25
MIN aggregate function 1-157

NEW expression 1-241
Non-matching index scans 1-165
Non-serializable objects 1-213
NOT boolean operator 1-223
NULL
not a data type 1-180
within method invocations 1-232
NULL expression 1-218
NUMERIC data type 1-204
Numeric data types 1-182
implicit conversion 1-184
promotion of in expressions 1-183
Numeric expressions 1-219
Numeric types
assigning and comparing to booleans 1-188

Objects
non-serializable 1-213
OCTET_LENGTH expression 1-219
OCTET_LENGTH function 1-159
oj (JDBC escape keyword) 6-38
ON
differences from USING in join expressions 1-85
ON clause
part of INNER JOIN expression 1-83
OR boolean operator 1-223
ORDER BY clause 1-93
affecting cursors 1-93
and Java data types 1-93
needed to guarantee order of results 1-107
Orderable Java data types 1-214
and indexes 1-42, 1-215
Ordering
not typically permitted with Java data types 1-214
Ordering Java data types 1-214
Outer joins 1-89
JDBC escape syntax for 6-38

password=userPassword (database connection URL attribute) 7-9
Precision
of decimal arithmetic expressions 1-185
Prepared statements
tracking of dependencies 1-15
use of dynamic parameters in 1-226
Primary key constraints 1-31
Primary keys
adding 1-29, 1-30
creating 1-29, 1-30
PROPERTIES clause
part of CREATE TABLE and CREATE INDEX statements 1-95
PropertyInfo class alias 1-40

Quantified comparison boolean operator 1-225

REAL data type 1-205
ReferencedColumnsDescriptor class alias 1-40
Referential integrity
via foreign keys 1-32
Reserved words 2-1–2-4
Resource manager
Cloudscape as in J2EE system 8-1
Result set node
part of statement execution plan 1-163
ResultSets
errors in statement affecting creation of 6-14
RIGHT OUTER JOIN expression 1-100
RowListVTI class alias 1-40
RTRIM (function) 1-160
RUNTIMESTATISTICS() function 1-161–1-168
accessing information from 1-162
and multi-threaded applications 1-167

Scalar subquery expression 1-218
ScalarSubquery 1-101
Scale
for decimal arithmetic 1-185
of decimal arithmetic expressions 1-185
SchemaName identifier 1-5
Schemas
APP as default 6-5
changing default for session 1-112
creating 1-44
dropping 1-68
Scroll insensitive cursors (JDBC) 6-27
Scroll insensitive cursors 6-27
<table>
<thead>
<tr>
<th>Index</th>
<th>IN-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second-chance type conversion</td>
<td>1-246</td>
</tr>
<tr>
<td>Select expression</td>
<td>1-102–1-105</td>
</tr>
<tr>
<td>SELECT statement</td>
<td>1-106–1-108</td>
</tr>
<tr>
<td>Serialization</td>
<td>1-212</td>
</tr>
<tr>
<td>SESSION_USER (function)</td>
<td>1-169</td>
</tr>
<tr>
<td>SET CONSTRAINTS statement</td>
<td>1-109</td>
</tr>
<tr>
<td>Set functions</td>
<td>1-129</td>
</tr>
<tr>
<td>SET LOCKING clause</td>
<td>1-48</td>
</tr>
<tr>
<td>SET RUNTIMESTATISTICS statement</td>
<td>1-111</td>
</tr>
<tr>
<td>SET SCHEMA statement</td>
<td>1-112</td>
</tr>
<tr>
<td>SET STATISTICS TIMING statement</td>
<td>1-113</td>
</tr>
<tr>
<td>SET TRANSACTION ISOLATION LEVEL statement</td>
<td>1-114</td>
</tr>
<tr>
<td>SET TRIGGERS statement</td>
<td>1-115</td>
</tr>
<tr>
<td>setAsciiStream</td>
<td>6-15</td>
</tr>
<tr>
<td>setBinaryStream</td>
<td>6-15</td>
</tr>
<tr>
<td>setUnicodeStream</td>
<td>6-15</td>
</tr>
<tr>
<td>setXXXStream requests</td>
<td>6-15</td>
</tr>
<tr>
<td>shutdown=true (database connection URL attribute)</td>
<td>7-10</td>
</tr>
<tr>
<td>SimpleColumnName identifier</td>
<td>1-7</td>
</tr>
<tr>
<td>SMALL INT data type</td>
<td>1-206</td>
</tr>
<tr>
<td>Special characters</td>
<td>1-2</td>
</tr>
<tr>
<td>escaping in SQL-J statements</td>
<td>1-2</td>
</tr>
<tr>
<td>SQL-92</td>
<td>1-2</td>
</tr>
<tr>
<td>features not supported by Cloudscape</td>
<td>3-1</td>
</tr>
<tr>
<td>features supported by Cloudscape</td>
<td>3-1</td>
</tr>
<tr>
<td>SQL-92 data types</td>
<td>1-2</td>
</tr>
<tr>
<td>supported by Cloudscape</td>
<td>1-80</td>
</tr>
<tr>
<td>SQL2Identifiers</td>
<td>1-3</td>
</tr>
<tr>
<td>definition</td>
<td>1-3</td>
</tr>
<tr>
<td>rules for 1-3–1-12</td>
<td></td>
</tr>
<tr>
<td>SQLExceptions</td>
<td>1-2</td>
</tr>
<tr>
<td>chaining 5-2, 6-23</td>
<td></td>
</tr>
<tr>
<td>importance of not catching within database-side methods</td>
<td>1-235</td>
</tr>
<tr>
<td>SQL-J identifiers</td>
<td>1-2</td>
</tr>
<tr>
<td>definition</td>
<td>1-3</td>
</tr>
<tr>
<td>SQLState values</td>
<td>5-1</td>
</tr>
<tr>
<td>SQLState XJZZZ</td>
<td>6-1</td>
</tr>
<tr>
<td>meaning of 6-1</td>
<td></td>
</tr>
<tr>
<td>Statement execution plan</td>
<td>1-163</td>
</tr>
<tr>
<td>StatementListVTI class alias</td>
<td>1-40</td>
</tr>
<tr>
<td>StatementName identifier</td>
<td>1-10</td>
</tr>
<tr>
<td>Storage properties</td>
<td>1-163</td>
</tr>
<tr>
<td>for indexes backing constraints</td>
<td>1-33</td>
</tr>
<tr>
<td>Stored prepared statements</td>
<td>1-18</td>
</tr>
<tr>
<td>altering</td>
<td>1-18</td>
</tr>
<tr>
<td>creating</td>
<td>1-45</td>
</tr>
<tr>
<td>recompiling invalid statements</td>
<td>1-18</td>
</tr>
<tr>
<td>Stream length specification not necessary</td>
<td>6-15</td>
</tr>
<tr>
<td>Streaming columns</td>
<td>1-246</td>
</tr>
<tr>
<td>as IN parameters to prepared statement</td>
<td>6-15</td>
</tr>
<tr>
<td>retrieving from ResultSets using getXXXStream methods</td>
<td>6-20</td>
</tr>
<tr>
<td>String expressions</td>
<td>1-246</td>
</tr>
<tr>
<td>with wildcards</td>
<td>1-220</td>
</tr>
<tr>
<td>Strings delimited by single quotation marks within SQL-J statements</td>
<td>1-2</td>
</tr>
<tr>
<td>SUBSTR (function)</td>
<td>1-172</td>
</tr>
<tr>
<td>SUBSTRING (function)</td>
<td>1-170</td>
</tr>
<tr>
<td>Subsubprotocol</td>
<td>1-246</td>
</tr>
<tr>
<td>part of database connection URL syntax</td>
<td>6-5</td>
</tr>
<tr>
<td>SUM aggregate function</td>
<td>1-174</td>
</tr>
<tr>
<td>SYSAliases system table</td>
<td>4-3</td>
</tr>
<tr>
<td>SYSCHECKS system table</td>
<td>4-5</td>
</tr>
<tr>
<td>SYSCOLUMNS system table</td>
<td>4-6</td>
</tr>
<tr>
<td>SYSCONGLOMERATES system table</td>
<td>4-9</td>
</tr>
<tr>
<td>SYSCONSTRAINTS system table</td>
<td>4-11</td>
</tr>
<tr>
<td>SYSDEPENDS system table</td>
<td>4-13</td>
</tr>
<tr>
<td>SYSErrors system table</td>
<td>4-15</td>
</tr>
<tr>
<td>SYSFILES system table</td>
<td>4-17</td>
</tr>
<tr>
<td>SYSFOREIGNKEYS system table</td>
<td>4-18</td>
</tr>
<tr>
<td>SYSJDBCTYPEINFO system table</td>
<td>4-19</td>
</tr>
<tr>
<td>SYSKEYS system table</td>
<td>4-20</td>
</tr>
<tr>
<td>SYSSCHEMAS system table</td>
<td>4-21</td>
</tr>
<tr>
<td>SYSTABLES system table</td>
<td>4-21</td>
</tr>
<tr>
<td>SYSTRIGGERS system table</td>
<td>4-24</td>
</tr>
<tr>
<td>SYSTVIEWS system table</td>
<td>4-26</td>
</tr>
<tr>
<td>t (JDBC escape keyword)</td>
<td>6-39</td>
</tr>
<tr>
<td>Table constraints</td>
<td>1-2</td>
</tr>
<tr>
<td>used in CREATE TABLE statement</td>
<td>1-29</td>
</tr>
<tr>
<td>Table subquery expression</td>
<td>1-218</td>
</tr>
<tr>
<td>TableExpression</td>
<td>1-116</td>
</tr>
<tr>
<td>Table-level constraints</td>
<td>1-29</td>
</tr>
<tr>
<td>TableName identifier</td>
<td>1-6</td>
</tr>
<tr>
<td>Tables</td>
<td>1-2</td>
</tr>
<tr>
<td>altering</td>
<td>1-20</td>
</tr>
<tr>
<td>changing lock granularity for</td>
<td>1-22</td>
</tr>
<tr>
<td>dropping</td>
<td>1-70</td>
</tr>
<tr>
<td>locking</td>
<td>1-91</td>
</tr>
<tr>
<td>specifying page size for</td>
<td>1-33</td>
</tr>
<tr>
<td>specifying properties for</td>
<td>1-33</td>
</tr>
<tr>
<td>TableSubquery</td>
<td>1-119</td>
</tr>
<tr>
<td>TIME data type</td>
<td>1-207</td>
</tr>
<tr>
<td>TIMESTAMP data type</td>
<td>1-208</td>
</tr>
<tr>
<td>TINYINT data type</td>
<td>1-209</td>
</tr>
</tbody>
</table>
TransactionListVTI class alias 1-40
TransactionTable class alias 1-40
TriggerName identifier 1-11
TriggerNewTransitionRows class alias 1-40
TriggerOldTransitionRows class alias 1-40
Triggers
and database events 1-52
and recursion 1-58
and the referencing clause 1-52
creating 1-51
defining when they fire 1-51
 disabling and reenabling 1-115
dropping 1-71
order of execution 1-54
referencing old and new values 1-52
statement vs. row triggers 1-53
the trigger action 1-54
transition row VTIs 1-56
trigger execution context 1-55
TRIM function 1-175
ts (JDBC escape keyword) 6-40
Type conversion of String types
implicit 1-186
Type correspondence
 between SQL-J and Java types 1-243–1-247
for JDBC and SQL-J types 6-25
TypeDescriptor class alias 1-40

U
Unicode
 setUnicodeStream 6-15
 support for 1-2
Unicode escapes
 support for in SQL-J statements 1-2
UNION 1-98
UNION ALL 1-98
Unique constraints 1-31
UPDATE statement 1-121
upgrade=true (database connection URL attribute) 7-12
UPPER function 1-177
USER function 1-178
user=userName (database connection URL attribute) 7-13
User-defined aggregates 1-179
dropping 1-64
User-defined data types 1-211–1-213
UserUtility class alias 1-40
USING
differences from ON in join expressions 1-85
performance implications of in join expressions 1-85
USING clause
 part of INNER JOIN expression 1-83

V
VALUES clause
 used to invoke work units 1-123
VALUES expression 1-123
Varbinary data types 1-201
VARCHAR data type 1-210
ViewName identifier 1-6
Views
 creating 1-60
 dropping 1-72
Virtual Table Interface (VTI) 1-76

W
WHERE clause
 SQL-J extensions to 1-125
WHERE CURRENT OF clause 1-127
Wildcards in SQL-J 1-2
Work units
 invoking 1-26, 1-123

X
XA 8-2