[vertical list of authors]

[cover art/text goes here]

Contents

Copyright

Second Edition (July 2005)

About this guide

For general information about the Derby documentation, such as a complete list of books, conventions, and further reading, see *Getting Started with Derby* .

Purpose of this document

This book, the *Derby Reference Manual* , provides reference information about Derby. It covers Derby's SQL language, the Derby implementation of JDBC, Derby system catalogs, Derby error messages, Derby properties, and SQL keywords.

Audience

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

Derby users who want a how-to approach to working with Derby or an introduction to Derby concepts should read the *Derby Developer's Guide* .

How this guide is organized

This guide includes the following sections:

- *SQL language reference*

  Reference information about Derby's SQL language, including manual pages for statements, functions, and other syntax elements.

- *SQL reserved words*

  SQL keywords beyond the standard SQL-92 keywords.

- *Derby support for SQL-92 features*

  A list of SQL-92 features that Derby does and does not support.

- *Derby System Tables*

  Reference information about the Derby system catalogs.

- *Derby exception messages and SQL states*

  Information about Derby exception messages.

- *JDBC Reference*

  Information about Derby's implementation of the JDBC interface include support for JDBC 2.0 features.

- *Setting attributes for the database connection URL*

  Information about the supported attributes to Derby's JDBC database connection URL.

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

  Information about the supported attributes to Derby's support for the Java Transaction API.

- *Derby API*

  Notes about proprietary APIs for Derby.

SQL language reference

Derby implements an SQL-92 core subset, as well as some SQL-99 features.

This section provides an overview of the current SQL language by describing the statements, built-in functions, data types, expressions, and special characters it contains.

Capitalization and special characters

Using the classes and methods of JDBC, you submit SQL statements to Derby as strings. The character set permitted for strings containing SQL statements is Unicode. 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. (In other words, a single quotation mark is the escape character for a single quotation mark.)

  A double quotation mark does not need an escape character. To represent a double quotation mark, simply use a double quotation mark. However, note that in a Java program, a double quotation mark requires the backslash escape character.

  Example:

```
-- a single quotation mark is the escape character
-- for a single quotation mark

VALUES 'Joe''s umbrella'
-- in ij, you don't need to escape the double quotation marks
VALUES 'He said, "hello!"'

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

- SQL 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*), unless they are delimited.
- Java-style identifiers are always case-sensitive.
- * is a wildcard within a *SelectExpression* . See The * wildcard . 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 (except when escaped with an escape character). See Boolean expression .
- 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.

SQL identifiers

An *identifier* is the representation within the language of items created by the user, as opposed to language keywords or commands. Some identifiers stand for *dictionary objects*, which are the objects you create- such as tables, views, indexes, columns, and constraints- that are stored in a database. They are called dictionary objects because Derby stores information about them in the system tables, sometimes known as a data dictionary. SQL also defines ways to alias these objects within certain statements.

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 *SQL92Identifier* s.

Rules for SQL92 identifiers

Ordinary identifiers are identifiers not surrounded by double quotation marks. Delimited identifiers are identifiers surrounded by double quotation marks.

An ordinary identifier must begin with a letter and contain only letters, underscore characters (_), and digits. The permitted letters and digits include all Unicode letters and digits, but Derby does not attempt to ensure that the characters in identifiers are valid in the database's locale.

A delimited identifier can contain any characters within the double quotation marks. The enclosing double 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). Derby 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 being represented).

So, in the following example:

```
"A.B"
```

is a dictionary object, while

```
"A"."B"
```

is a dictionary object qualified by another dictionary object (such as a column named "B" within the table "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 by double quotes), because they are automatically translated into uppercase by the system. You cannot use reserved words as identifiers for dictionary objects unless they are delimited. If you attempt to use a name longer than 128 characters, *SQLException* X0X11 is raised.

Derby defines keywords beyond those specified by the SQL-92 standard (see SQL reserved words).

Example

```
-- the view name is stored in the
-- system catalogs as ANIDENTIFIER
CREATE VIEW AnIdentifier (RECEIVED) AS VALUES 1
-- the view name is stored in the system
-- catalogs with case intact
CREATE VIEW "ACaseSensitiveIdentifier" (RECEIVED) AS VALUES 1
```

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

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.

column-Name

In many places in the SQL syntax, you can represent the name of a column by qualifying it with a *table-Name* or *correlation-Name*.

In some situations, you cannot qualify a *column-Name* with a *table-Name* or a *correlation-Name,* but must use a *Simple-column-Name* 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*)

You cannot use correlation-Names for updatable columns; using correlation-Names in this way will cause an SQL exception. For example:

```
SELECT c11 AS col1, c12 AS col2, c13 FROM t1 FOR UPDATE of c11,c13
```

In this example, the correlation-Name `col1 FOR c11` is not permitted because `c11` is listed in the FOR UPDATE list of columns. You can use the correlation-Name `FOR c12` because it is not in the FOR UPDATE list.

Syntax

```
[ {
table-Name
 |
correlation-Name
 } . ]
SQL92Identifier
```

Example

```
-- C.Country is a column-Name qualified with a
--
correlation-Name
.
SELECT C.Country
FROM APP.Countries C
```

correlation-Name

A *correlation-Name* is given to a table expression in a FROM clause as a new name or alias for that table. You do not qualify a *correlation-Name* with a *schema-Name*.

You cannot use correlation-Names for updatable columns; using correlation-Names in this way will cause an SQL exception. For example:

```
SELECT c11 AS col1, c12 AS col2, c13 FROM t1 FOR UPDATE of c11,c13
```

In this example, the correlation-Name `col1 FOR c11` is not permitted because `c11` is listed in the FOR UPDATE list of columns. You can use the correlation-Name `FOR c12` because it is not in the FOR UPDATE list.

Syntax

```
SQL92Identifier
```

Example

```
-- C is a correlation-Name
SELECT C.NAME
FROM SAMP.STAFF C
```

new-table-Name

A *new-table-Name* represents a renamed table. You cannot qualify a *new-table-Name* with a *schema-Name*.

Syntax

```
SQL92Identifier
```

Example

```
-- FlightBooks is a new-table-Name that does not include a schema-Name
RENAME TABLE FLIGHTAVAILABILITY TO FLIGHTAVAILABLE
```

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 cannot create objects in schemas starting with SYS.

Thus, you can qualify references to tables with the schema name. When a schemaName 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 Derby System Tables .

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

Syntax

```
SQL92Identifier
```

Example

```
-- SAMP.EMPLOYEE is a table-Name qualified by a schemaName
SELECT COUNT(*) FROM SAMP.EMPLOYEE
-- You must qualify system catalog names with their schema, SYS
SELECT COUNT(*) FROM SYS.SysColumns
```

Simple-column-Name

A *Simple-column-Name* is used to represent a column when it cannot be qualified by a *table-Name* or *correlation-Name*. This is the case when the qualification is fixed, as it is in a column definition within a CREATE TABLE statement.

Syntax

```
SQL92Identifier
```

Example

```
-- country is a Simple-column-Name
CREATE TABLE CONTINENT (COUNTRY VARCHAR(26) NOT NULL PRIMARY KEY,
COUNTRY_ISO_CODE CHAR(2), REGION VARCHAR(26))
```

synonym-Name

A *synonym-Name* represents a synonym for a table or a view. You can qualify a *synonym-Name* with a *schema-Name*.

Syntax

```
[
schemaName
. ]
SQL92Identifier
```

table-Name

A *table-Name* represents a table. You can qualify a *table-Name* with a *schemaName*.

Syntax

```
[
schemaName
. ]
```

```
SQL92Identifier
```

Example

```
-- SAMP.PROJECT is a table-Name that includes a schemaName
SELECT COUNT(*) FROM SAMP.PROJECT
```

view-Name

A *view-Name* represents a table or a view. You can qualify a *view-Name* with a *schema-Name*.

Syntax

```
[
schemaName
. ]
SQL92Identifier
```

Example

```
-- This is a View qualified by a schema-Name
SELECT COUNT(*) FROM SAMP.EMP_RESUME
```

index-Name

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

Syntax

```
[
schemaName
 . ]
SQL92Identifier
```

Example

```
DROP INDEX APP.ORIGINDEX;
-- OrigIndex is an index-Name without a schema-Name
CREATE INDEX ORIGINDEX ON FLIGHTS (ORIG_AIRPORT)
```

constraint-Name

You cannot qualify constraint-names.

Syntax

```
SQL92Identifier
```

Example

```
-- country_fk2 is a constraint name
CREATE TABLE DETAILED_MAPS (COUNTRY_ISO_CODE CHAR(2)
CONSTRAINT country_fk2 REFERENCES COUNTRIES)
```

cursor-Name

A *cursor-Name* refers to a cursor. No SQL 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 the *Derby Developer's Guide* . If you assign a name to a cursor, you can refer to that name from within SQL statements.

You cannot qualify a *cursor-Name*.

Syntax

```
SQL92Identifier
```

Example

```
stmt.executeUpdate("UPDATE SAMP.STAFF SET COMM = " +
"COMM + 20 " + "WHERE CURRENT OF " + ResultSet.getCursorName());
```

TriggerName

A *TriggerName* refers to a trigger created by a user.

Syntax

```
[
schemaName
 . ]
SQL92Identifier
```

Example

```
DROP TRIGGER TRIG1
```

AuthorizationIdentifier

User names within the Derby system are known as *authorization identifiers*. The authorization identifier represents the name of the user, if one has been provided in the connection request. The default schema for a user is equal to its authorization identifier. User names can be case-sensitive within the authentication system, but they are always case-insensitive within Derby's authorization system unless they are delimited. For more information, see the *Derby Developer's Guide* .

Syntax

```
SQL92Identifier
```

Example

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'Amber,FRED')
```

Statements

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.

The sections provide general information about statement use, and descriptions of the specific statements.

Interaction with the dependency system

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

Prepared statements depend on the dictionary objects and statements they reference. (Dictionary objects include tables, columns, constraints, indexes, views, and triggers.) Removing or modifying the dictionary objects or statements on which they depend invalidates them internally, which means that Derby 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 Derby 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 Derby 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
-- this example uses the ij command prepare,
-- which prepares a statement
ij> prepare p1 AS 'INSERT INTO MyTable VALUES (4)';
-- p1 depends on mytable;
ij> execute p1;
1 row inserted/updated/deleted
-- Derby  executes it without recompiling
ij> CREATE INDEX i1 ON mytable(mycol);
0 rows inserted/updated/deleted
-- p1 is temporarily invalidated because of new index
ij> execute p1;
1 row inserted/updated/deleted
-- Derby  automatically recompiles p1 and executes it
ij> DROP TABLE mytable;
0 rows inserted/updated/deleted
-- Derby  permits you to drop table
-- because result set of p1 is closed
-- however, the statement p1 is temporarily invalidated
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> execute p1;
1 row inserted/updated/deleted
-- Because p1 is invalid, Derby  tries to recompile it
-- before executing.
-- It is successful and executes.
ij> DROP TABLE mytable;
0 rows inserted/updated/deleted
-- statement p1 is now invalid,
-- and this time the attempt to recompile it
-- upon execution will fail
ij> execute p1;
ERROR 42X05: Table 'MYTABLE' does not exist.
```

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
- increase the width of a VARCHAR, CHAR VARYING, and CHARACTER VARYING column
- override row-level locking for the table (or drop the override)

Syntax

```
ALTER TABLE
table-Name

{
    ADD COLUMN
column-definition
 |
    ADD
```

```
CONSTRAINT clause

 |
    DROP { PRIMARY KEY | FOREIGN KEY constraint-name | UNIQUE
         constraint-name | CHECK constraint-name | CONSTRAINT
constraint-name }
    ALTER

column-alteration

 |
    LOCKSIZE { ROW | TABLE }
}
```

column-definition

```
Simple-column-Name

DataType
[

Column-level-constraint

 ]*
[ [ WITH ] DEFAULT {ConstantExpression | NULL } ]
```

column-alteration

```
column-Name SET DATA TYPE VARCHAR(integer) |
column-name SET INCREMENT BY integer-constant
```

In the column-alteration, SET INCREMENT BY integer-constant, specifies the interval between consecutive values of the identity column. The next value to be generated for the identity column will be determined from the last assigned value with the increment applied. The column must already be defined with the IDENTITY attribute.

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 *column-definition* 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 if you give a default value; otherwise, an exception is thrown when the ALTER TABLE statement is executed.

Just as in CREATE TABLE, if the column definition includes a unique or primary key constraint, the column cannot contain null values, so the NOT NULL attribute must also be specified (SQLSTATE 42831).

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:

- When adding a foreign key or check constraint to an existing table, Derby checks

the table to make sure existing rows satisfy the constraint. If any row is invalid, Derby throws a statement exception and the constraint is not added.

- All columns included in a primary key must contain non null data and be unique.

  ALTER TABLE ADD UNIQUE or PRIMARY KEY provide a shorthand method of defining a primary key composed of a single column. If PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause were specified as a separate clause. The column cannot contain null values, so the NOT NULL attribute must also be specified.

For information on the syntax of constraints, see CONSTRAINT clause . 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 the physical index that enforces the constraint (also known as a *backing index*).

Modifying columns

The *column-alteration* allows you to alter the named column in the following ways:

- Increasing the length of an existing VARCHAR column. CHARACTER VARYING or CHAR VARYING can be used as synonyms for the VARCHAR keyword.

  To increase the width of a column of these types, specify the data type and new size after the column name.

  You are not allowed to decrease the width or to change the data type. You are not allowed to increase the width of a column that is part of a primary or unique key referenced by a foreign key constraint or that is part of a foreign key constraint.

- Specifying the interval between consecutive values of the identity column.

  To set an interval between consecutive values of the identity column, specify the integer-constant. You must previously define the column with the IDENTITY attribute (SQLSTATE 42837). If there are existing rows in the table, the values in the column for which the SET INCREMENT default was added do not change.

Setting defaults

You can specify a default value for a new 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.

For more information about defaults, see CREATE TABLE statement .

Changing the lock granularity for the table

The LOCKSIZE clause allows you to override row-level locking for the specific table, if your system uses the default setting of row-level locking. (If your system is set for table-level locking, you cannot change the locking granularity to row-level locking, although Derby allows you to use the LOCKSIZE 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 LOCKSIZE clause in the ALTER TABLE STATEMENT. For information about why this is sometimes useful, see *Tuning Derby* .

Examples

```
-- Add a new column with a column-level constraint
-- to an existing table
-- An exception will be thrown if the table
-- contains any rows
-- 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 CHECK (REGION IS NOT NULL);

-- Add a new unique constraint to an existing table
-- An exception will be thrown if duplicate keys are found
ALTER TABLE SAMP.DEPARTMENT
ADD CONSTRAINT NEW_UNIQUE UNIQUE (DEPTNO);

-- add a new foreign key constraint to the
-- Cities table. Each row in Cities is checked
-- to make sure it satisfied the constraints.
-- if any rows don't satisfy the constraint, the
-- constraint is not added
ALTER TABLE CITIES ADD CONSTRAINT COUNTRY_FK
Foreign Key (COUNTRY) REFERENCES COUNTRIES (COUNTRY);

-- Add a primary key constraint to a table
-- First, create a new table
CREATE TABLE ACTIVITIES (CITY_ID INT NOT NULL,
SEASON CHAR(2), ACTIVITY VARCHAR(32) NOT NULL);
-- You will not be able to add this constraint if the
-- columns you are including in the primary key have
-- null data or duplicate values.
ALTER TABLE Activities ADD PRIMARY KEY (city_id, activity);

-- Drop a primary key constraint from the CITIES table

ALTER TABLE Cities DROP CONSTRAINT Cities_PK;
-- Drop a foreign key constraint from the CITIES table
ALTER TABLE Cities DROP CONSTRAINT COUNTRIES_FK;
-- add a DEPTNO column with a default value of 1
ALTER TABLE SAMP.EMP_ACT ADD COLUMN DEPTNO INT DEFAULT 1;
-- increase the width of a VARCHAR column
ALTER TABLE SAMP.EMP_PHOTO ALTER PHOTO_FORMAT SET DATA TYPE VARCHAR(30);
-- change the lock granularity of a table
ALTER TABLE SAMP.SALES LOCKSIZE TABLE;
```

Results

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.

CREATE statements

Use the Create statements with functions, indexes, procedures, schemas, synonyms, tables, triggers, and views.

CREATE FUNCTION statement

The CREATE FUNCTION statement allows you to create Java functions, which you can then use in an expression.

Syntax

```
CREATE FUNCTION
function-name
 ( [
FunctionParameter
   [,
FunctionParameter
] ] * ) RETURNS DataType [
FunctionElement
 ] *
```

function-Name

```
[
schemaName
. ]
SQL92Identifier
```

If schema-Name is not provided, the current schema is the default schema. If a qualified procedure name is specified, the schema name cannot begin with SYS.

FunctionParameter

```
[ parameter-Name ] DataType
```

PararameterName must be unique within a function.

The syntax of *DataType* is described in Data types .

Note: Long data-types such as LONG VARCHAR, LONG VARCHAR FOR BIT DATA, CLOB, and BLOB are not allowed as parameters in a CREATE FUNCTION statement.

FunctionElement

```
 {
| LANGUAGE { JAVA }
| EXTERNAL NAME string
| PARAMETER STYLE JAVA
| { NO SQL | CONTAINS SQL | READS SQL DATA }
| { RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT }
 }
```

LANGUAGE

JAVA- the database manager will call the function as a public static method in a Java class.

EXTERNAL NAME *string*
String describes the Java method to be called when the function is executed, and takes the following form:

```
class_name.method_name
```

The External Name cannot have any extraneous spaces.

PARAMETER STYLE

JAVA - The function will use a parameter-passing convention that conforms to the Java language and SQL Routines specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values. Result sets are returned through additional parameters to the Java method of type java.sql.ResultSet[] that are passed single entry arrays.

Derby does not support long column types (for example Long Varchar, BLOB, and so on). An error will occur if you try to use one of these long column types.

NO SQL, CONTAINS SQL, READS SQL DATA

Indicates whether the function issues any SQL statements and, if so, what type.

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the function. Statements that are not supported in any function return a different error.

NO SQL
Indicates that the function cannot execute any SQL statements

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the function. Statements that are not supported in any stored function return a different error. This is the default value.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
Specifies whether the function is called if any of the input arguments is null. The result is the null value.

RETURNS NULL ON NULL INPUT
Specifies that the function is not invoked if any of the input arguments is null. The result is the null value.

CALLED ON NULL INPUT
Specifies that the function is invoked if any or all input arguments are null. This specification means that the function must be coded to test for null argument values. The function can return a null or non-null value. This is the default setting.

The function elements may appear in any order, but each type of element can only appear once. A function definition must contain these elements:

- **LANGUAGE**
- **PARAMETER STYLE**
- **EXTERNAL NAME**

Example

```
CREATE FUNCTION TO_DEGREES(RADIANS DOUBLE) RETURNS DOUBLE
PARAMETER STYLE JAVA NO SQL LANGUAGE JAVA
EXTERNAL NAME 'java.lang.Math.toDegrees'
```

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] INDEX
index-Name

ON
table-Name
 (
Simple-column-Name
 [ ASC | DESC ]
    [ ,
Simple-column-Name
 [ ASC | DESC ]] * )
```

The maximum number of columns for an index key in Derby is 16.

An index name cannot exceed 128 characters.

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

Derby can use indexes to improve the performance of data manipulation statements (see *Tuning Derby*). In addition, 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, but Derby does not.) 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.

By default, Derby uses the ascending order of each column to create the index. Specifying ASC after the column name does not alter the default behavior. The DESC keyword after the column name causes Derby to use descending order for the column to create the index. Using the descending order for a column can help improve the performance of queries that require the results in mixed sort order or descending order and for queries that select the minimum or maximum value of an indexed column.

If a qualified index name is specified, the schema name cannot begin with SYS.

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 can not create an index on those columns. Derby 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. For example, to find out the name of the index that backs a constraint called FLIGHTS_PK:

```
SELECT CONGLOMERATENAME FROM SYS.SYSCONGLOMERATES,
SYS.SYSCONSTRAINTS WHERE
SYS.SYSCONGLOMERATES.TABLEID = SYSCONSTRAINTS.TABLEID
AND CONSTRAINTNAME = 'FLIGHTS_PK'
```

```
CREATE INDEX OrigIndex ON Flights(orig_airport);
-- money is usually ordered from greatest to least,
-- so create the index using the descending order
CREATE INDEX PAY_DESC ON SAMP.EMPLOYEE (SALARY);
-- use a larger page size for the index
call
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.storage.pageSize','8192');
CREATE INDEX IXSALE ON SAMP.SALES (SALES);
call
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.storage.pageSize',NULL);
```

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 existing row in a table is larger than half the page size of the index, creating an index on those key columns for the table will fail. This error only occurs when creating an index if an existing row in the table fails the criteria. After an index is created, inserts may fail if the size of their associated key exceeds the criteria.

Statement 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 PROCEDURE statement

The CREATE PROCEDURE statement allows you to create Java stored procedures, which you can then call using the CALL PROCEDURE statement.

Syntax

```
CREATE PROCEDURE
procedure-Name
 ( [
ProcedureParameter
   [,
ProcedureParameter
] ] * )
[
ProcedureElement
 ] *
```

procedure-Name

```
[
schemaName
. ]
SQL92Identifier
```

If schema-Name is not provided, the current schema is the default schema. If a qualified procedure name is specified, the schema name cannot begin with SYS.

ProcedureParameter

```
[ { IN | OUT | INOUT } ] [ parameter-Name ] DataType
```

The default value for a parameter is IN. ParameterName must be unique within a procedure.

The syntax of *DataType* is described in Data types .

Note: Long data-types such as LONG VARCHAR, LONG VARCHAR FOR BIT DATA, CLOB, and BLOB are not allowed as parameters in a CREATE PROCEDURE statement.

ProcedureElement

```
 {
| [ DYNAMIC ] RESULT SETS
INTEGER

| LANGUAGE { JAVA }
| EXTERNAL NAME string
| PARAMETER STYLE JAVA
| { NO SQL | MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA }
 }
```

DYNAMIC RESULT SETS *integer*

Indicates the estimated upper bound of returned result sets for the procedure. Default is no (zero) dynamic result sets.

LANGUAGE

JAVA- the database manager will call the procedure as a public static method in a Java class.

EXTERNAL NAME *string*
String describes the Java method to be called when the procedure is executed, and takes the following form:

```
class_name.method_name
```

The External Name cannot have any extraneous spaces.

PARAMETER STYLE

JAVA - The procedure will use a parameter-passing convention that conforms to the Java language and SQL Routines specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values. Result sets are returned through additional parameters to the Java method of type java.sql.ResultSet [] that are passed single entry arrays.

Derby does not support long column types (for example Long Varchar, BLOB, and so on). An error will occur if you try to use one of these long column types.

NO SQL, CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA

Indicates whether the stored procedure issues any SQL statements and, if so, what type.

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the stored procedure. Statements that are not supported in any stored procedure return a different error. MODIFIES SQL DATA is the default value.

NO SQL
Indicates that the stored procedure cannot execute any SQL statements

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the stored procedure. Statements that are not supported in any stored procedure return a different error.

MODIFIES SQL DATA
Indicates that the stored procedure can execute any SQL statement except statements that are not supported in stored procedures.

The procedure elements may appear in any order, but each type of element can only appear once. A procedure definition must contain these elements:

- **LANGUAGE**
- **PARAMETER STYLE**
- **EXTERNAL NAME**

Example

```
CREATE PROCEDURE SALES.TOTAL_REVENUE(IN S_MONTH INTEGER,
IN S_YEAR INTEGER, OUT TOTAL DECIMAL(10,2))
PARAMETER STYLE JAVA READS SQL DATA LANGUAGE JAVA EXTERNAL NAME
'com.acme.sales.calculateRevenueByMonth'
```

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. A schema name cannot exceed 128 characters. Schema names must be unique within the database.

```
-- Create a schema for employee-related tables
CREATE SCHEMA EMP;
-- 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) NOT NULL, SEGMENT_NUMBER INT NOT NULL,
        FLIGHT_DATE DATE NOT NULL, ECONOMY_SEATS_TAKEN INT,
        BUSINESS_SEATS_TAKEN INT, FIRSTCLASS_SEATS_TAKEN INT,
        CONSTRAINT FLT_AVAIL_PK
        PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER, FLIGHT_DATE));

CREATE TABLE EMP.AVAILABILITY
        (HOTEL_ID INT NOT NULL, BOOKING_DATE DATE NOT NULL, ROOMS_TAKEN
INT,
        CONSTRAINT HOTELAVAIL_PK PRIMARY KEY (HOTEL_ID, BOOKING_DATE));
```

CREATE SYNONYM statement

Use the CREATE SYNONYM statement to provide an alternate name for a table or a view that is present in the same schema or another schema. You can also create synonyms for other synonyms, resulting in nested synonyms. A synonym can be used instead of the original qualified table or view name in SELECT, INSERT, UPDATE, DELETE or LOCK TABLE statements. You can create a synonym for a table or a view that doesn't exist, but the target table or view must be present before the synonym can be used.

Synonyms share the same namespace as tables or views. You cannot create a synonym with the same name as a table that already exists in the same schema. Similarly, you cannot create a table or view with a name that matches a synonym already present.

A synonym can be defined for a table/view that does not exist when you create the synonym. If the table or view doesn't exist, you will receive a warning message (SQLSTATE 01522). The referenced object must be present when you use a synonym in a DML statement.

You can create a nested synonym (a synonym for another synonym), but any attempt to create a synonym that results in a circular reference will return an error message (SQLSTATE 42916).

Synonyms cannot be defined in system schemas. All schemas starting with 'SYS' are considered system schemas and are reserved by Derby.

A synonym cannot be defined on a temporary table. Attempting to define a synonym on a temporary table will return an error message (SQLSTATE XCL51).

Syntax

```
CREATE SYNONYM
synonym-Name
 FOR {
view-Name
 |
table-Name
 }
```

The synonym-Name in the statement represents the synonym name you are giving the target table or view, while the view-Name or table-Name represents the original name of the target table or view.

Example

CREATE SYNONYM SAMP.T1 FOR SAMP.TABLEWITHLONGNAME

CREATE TABLE statement

A CREATE TABLE statement creates a table. Tables contain columns and constraints, rules to which data must conform. Table-level constraints specify a column or columns. Columns have a data type and can specify column constraints (column-level constraints).

For information about constraints, see CONSTRAINT clause .

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. See Column default .

You can specify storage properties such as page size for a table by calling the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure.

If a qualified table name is specified, the schema name cannot begin with *SYS*.

Syntax

```
CREATE TABLE
table-Name

    ( {
column-definition
 |
Table-level constraint
}
    [ , {
column-definition
 |
Table-level constraint
} ] * )
```

Example

```
CREATE TABLE HOTELAVAILABILITY
     (HOTEL_ID INT NOT NULL, BOOKING_DATE DATE NOT NULL,
        ROOMS_TAKEN INT DEFAULT 0, PRIMARY KEY (HOTEL_ID, BOOKING_DATE));
-- the table-level primary key definition allows you to
-- include two columns in the primary key definition
PRIMARY KEY (hotel_id, booking_date))
-- assign an identity column attribute to an INTEGER
-- column, and also define a primary key constraint
-- on the column
CREATE TABLE PEOPLE
        (PERSON_ID INT NOT NULL GENERATED ALWAYS AS IDENTITY
        CONSTRAINT PEOPLE_PK PRIMARY KEY, PERSON VARCHAR(26));
```

```
-- assign an identity column attribute to a SMALLINT
-- column with an initial value of 5 and an increment value
-- of 5.
CREATE TABLE GROUPS
        (GROUP_ID SMALLINT NOT NULL GENERATED ALWAYS AS IDENTITY
        (START WITH 5, INCREMENT BY 5), ADDRESS VARCHAR(100), PHONE
VARCHAR(15));
```

Note: For more examples of CREATE TABLE statements using the various constraints, see CONSTRAINT clause .

column-definition:

```
Simple-column-Name

DataType
    [
Column-level-constraint
 ]*
    [ [ WITH ] DEFAULT { ConstantExpression | NULL }
        |
generated-column-spec
 ]
    [
Column-level-constraint
 ]*
```

The syntax of *Data-Type* is described in Data types .

The syntaxes of *Column-level-constraint* and *Table-level constraint* are described in CONSTRAINT clause .

Column default

For the definition of a default value, a *ConstantExpression* is an expression that does not refer to any table. It can include constants, date-time special registers, current schemas, users, and null.

generated-column-spec:

```
[ GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ ( START WITH IntegerConstant
[ ,INCREMENT BY IntegerConstant] ) ]  ]  ]
```

Identity column attributes

For SMALLINT, INT, and BIGINT columns with identity attributes, Derby automatically assigns increasing integer values to the column. Identity column attributes behave like other defaults in that when an insert statement does not specify a value for the column, Derby automatically provides the value. However, the value is not a constant; Derby automatically increments the default value at insertion time.

The IDENTITY keyword can only be specified if the data type associated with the column is one of the following exact integer types.

- SMALLINT
- INT
- BIGINT

There are two kinds of identity columns in Derby: those which are GENERATED

ALWAYS and those which are GENERATED BY DEFAULT.

GENERATED ALWAYS

An identity column that is GENERATED ALWAYS will increment the default value on every insertion and will store the incremented value into the column. Unlike other defaults, you cannot insert a value directly into or update an identity column that is GENERATED ALWAYS. Instead, either specify the DEFAULT keyword when inserting into the identity column, or leave the identity column out of the insertion column list altogether. For example:

```
create table greetings
        (i int generated always as identity, ch char(50));
insert into greetings values (DEFAULT, 'hello');
insert into greetings(ch) values ('bonjour');
```

Automatically generated values in a GENERATED ALWAYS identity column are unique. Creating an identity column does not create an index on the column.

GENERATED BY DEFAULT

An identity column that is GENERATED BY DEFAULT will only increment and use the default value on insertions when no explicit value is given. Unlike GENERATED ALWAYS columns, you can specify a particular value in an insertion statement to be used instead of the generated default value.

To use the generated default, either specify the DEFAULT keyword when inserting into the identity column, or just leave the identity column out of the insertion column list. To specify a value, included it in the insertion statement. For example:

```
create table greetings
        (i int generated by default as identity, ch char(50));
-- specify value "1":
insert into greetings values (1, 'hi');
-- use generated default
insert into greetings values (DEFAULT, 'salut');
-- use generated default
insert into greetings(ch) values ('bonjour');
```

Note that unlike a GENERATED ALWAYS column, a GENERATED BY DEFAULT column does not guarantee uniqueness. Thus, in the above example, the `hi` and `salut` rows will both have an identity value of "1", because the generated column starts at "1" and the user-specified value was also "1". To prevent duplication, especially when loading or importing data, create the table using the START WITH value which corresponds to the first identity value that the system should assign. To check for this condition and disallow it, you can use a primary key or unique constraint on the GENERATED BY DEFAULT identity column.

By default, the initial value of an identity column is 1, and the amount of the increment is 1. You can specify non-default values for both the initial value and the interval amount when you define the column with the key words STARTS WITH and INCREMENT BY. And if you specify a negative number for the increment value, Derby *decrements* the value with each insert. If this value is 0, or positive, Derby increments the value with each insert.

The maximum and minimum values allowed in identity columns are determined by the data type of the column. Attempting to insert a value outside the range of values supported by the data type raises an exception.

Table1. Maximum and Minimum Values for Columns with Generated Column Specs

Data type	Maximum Value	Minimum Value
SMALLINT	32767 (*java.lang.Short.MAX_VALUE*)	-32768 (*java.lang.Short.MIN_VALUE*)
INT	2147483647 (*java.lang.Integer.MAX_VALUE*)	-2147483648 (*java.lang.Integer.MIN_VALUE*)
BIGINT	9223372036854775807 (*java.lang.Long.MAX_VALUE*)	-9223372036854775808 (*java.lang.Long.MIN_VALUE*)

Automatically generated values in an identity column are unique. Use a primary key or unique constraint on a column to guarantee uniqueness. Creating an identity column *does not* create an index on the column.

The `IDENTITY_VAL_LOCAL` function is a non-deterministic function that returns the most recently assigned value for an identity column. See IDENTITY_VAL_LOCAL for more information.

Note: Specify the schema, table, and column name using the same case as those names are stored in the system tables--that is, all upper case unless you used delimited identifiers when creating those database objects.

Derby keeps track of the last increment value for a column in a cache. It also stores the value of what the next increment value will be for the column on disk in the *AUTOINCREMENTVALUE* column of the *SYS.SYSCOLUMNS* system table. Rolling back a transaction does not undo this value, and thus rolled-back transactions can leave "gaps" in the values automatically inserted into an identity column. Derby behaves this way to avoid locking a row in *SYS.SYSCOLUMNS* for the duration of a transaction and keeping concurrency high.

When an insert happens within a triggered-SQL-statement, the value inserted by the triggered-SQL-statement into the identity column is available from *ConnectionInfo* only within the trigger code. The trigger code is also able to see the value inserted by the statement that caused the trigger to fire. However, the statement that caused the trigger to fire is not able to see the value inserted by the triggered-SQL-statement into the identity column. Likewise, triggers can be nested (or recursive). An SQL statement can cause trigger T1 to fire. T1 in turn executes an SQL statement that causes trigger T2 to fire. If both T1 and T2 insert rows into a table that cause Derby to insert into an identity column, trigger T1 cannot see the value caused by T2's insert, but T2 can see the value caused by T1's insert. Each nesting level can see increment values generated by itself and previous nesting levels, all the way to the top-level SQL statement that initiated the recursive triggers. You can only have 16 levels of trigger recursion.

Example

```
create table greetings
  (i int generated by default as identity (START WITH 2, INCREMENT BY 1),
  ch char(50));
-- specify value "1":
insert into greetings values (1, 'hi');
-- use generated default
insert into greetings values (DEFAULT, 'salut');
-- use generated default
insert into greetings(ch) values ('bonjour');
```

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 one that starts with *SYS*. The trigger need not reside in the same schema as the table on which it is defined.

If a qualified trigger name is specified, the schema name cannot begin with *SYS*.

Syntax

```
CREATE TRIGGER
TriggerName

{ AFTER | NO CASCADE BEFORE }
{ INSERT | DELETE | UPDATE } [ OF column-Name [,
column-Name
]* ]
ON
table-Name

[
ReferencingClause
 ]
FOR EACH { ROW | STATEMENT } MODE DB2SQL

Triggered-SQL-statement
```

Before or after: when triggers fire
Triggers are defined as either *Before* or *After* triggers.

- *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 versus row triggers).
- *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 versus row triggers).

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 (see Syntax above):

- 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 triggered-SQL-statements need to refer to data that is currently being changed by the database event that caused them to fire. The triggered-SQL-statement might need to refer to the new (post-change or "after") values.

Derby 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. Changed data can be referred to in the triggered-SQL-statement using *transition variables* or *transition tables*. The referencing clause allows you to provide a correlation name or alias for these transition variables by specifying OLD/NEW AS *correlation-Name* .

For example, if you add the following clause to the trigger definition:

```
REFERENCING OLD AS DELETEDROW
```

you can then refer to this correlation name in the triggered-SQL-statement:

```
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 versus row triggers) 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 triggered-SQL-statement or the trigger qualification. The referencing clause allows you to provide a correlation name or alias for these transition tables by specifying OLD_TABLE/NEW_TABLE AS correlation-Name

For example:

```
REFERENCING OLD_TABLE AS DeletedHotels
```

allows you to use that new identifier (*DeletedHotels*) in the triggered-SQL-statement:

```
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 versus row triggers) 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.

Statement versus row triggers
You must specify whether a trigger is a *statement trigger* or a *row trigger*:

- *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.

Triggered-SQL-statement
The action defined by the trigger is called the triggered-SQL-statement (in Syntax above, see the last line). 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 CALL statement.
- Before triggers cannot have INSERT, UPDATE or DELETE statements as their action.

The triggered-SQL-statement can 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.

For more information on triggered-SQL-statements, see the *Derby Developer's Guide* .

Order of execution
When a database event occurs that fires a trigger, Derby performs actions in this order:

- It fires *No Cascade Before* triggers.
- 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.

```
-- Statements and triggers:

CREATE TRIGGER t1_NO_CASCADE BEFORE UPDATE ON x
  FOR EACH ROW MODE DB2SQL
  values app.notifyEmail('Jerry', 'Table x is about to be updated');

CREATE TRIGGER FLIGHTSDELETE
  AFTER DELETE ON FLIGHTS
  REFERENCING OLD_TABLE AS DELETEDFLIGHTS
  FOR EACH STATEMENT MODE DB2SQL
  DELETE FROM FLIGHTAVAILABILITY WHERE FLIGHT_ID IN
  (SELECT FLIGHT_ID FROM DELETEDFLIGHTS);

CREATE TRIGGER FLIGHTSDELETE3
  AFTER DELETE ON FLIGHTS
  REFERENCING OLD AS OLD
  FOR EACH ROW MODE DB2SQL
  DELETE FROM FLIGHTAVAILABILITY WHERE FLIGHT_ID = OLD.FLIGHT_ID;
```

Note: You can find more examples in the *Derby Developer's Guide* .

Trigger recursion

The maximum trigger recursion depth is 16.

Related information
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

ReferencingClause:

```
REFERENCING
{
{ OLD | NEW } [ AS ] correlation-Name [ { OLD | NEW } [ AS ]
correlation-Name ] |
{ OLD_TABLE | NEW_TABLE } [ AS ] Identifier [ { OLD_TABLE | NEW_TABLE }
```

```
[AS] Identifier ]
}
```

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.

If a qualified view name is specified, the schema name cannot begin with *SYS*.

Syntax

```
CREATE VIEW
view-Name

    [ (
Simple-column-Name
 [,
Simple-column-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. All columns in a view must be uniquely named.

```
CREATE VIEW SAMP.V1 (COL_SUM, COL_DIFF)
        AS SELECT COMM + BONUS, COMM - BONUS
        FROM SAMP.EMPLOYEE;

CREATE VIEW SAMP.VEMP_RES (RESUME)
        AS VALUES 'Delores M. Quintana', 'Heather A. Nicholls', 'Bruce
Adamson';

CREATE VIEW SAMP.PROJ_COMBO
        (PROJNO, PRENDATE, PRSTAFF, MAJPROJ)
        AS SELECT PROJNO, PRENDATE, PRSTAFF, MAJPROJ
        FROM SAMP.PROJECT UNION ALL
SELECT PROJNO, EMSTDATE, EMPTIME, EMPNO
        FROM SAMP.EMP_ACT
        WHERE EMPNO IS NOT NULL;
```

Statement dependency system
View definitions are dependent on the tables and views 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. Statements that reference the view depend on indexes the view uses; which index a view uses can change from statement to statement based on how the query is optimized. For example, given:

```
CREATE TABLE T1 (C1 DOUBLE PRECISION);

CREATE FUNCTION SIN (DATA DOUBLE)
        RETURNS DOUBLE EXTERNAL NAME 'java.lang.Math.sin'
        LANGUAGE JAVA PARAMETER STYLE JAVA;

CREATE VIEW V1 (C1) AS SELECT SIN(C1) FROM T1;
```

the following SELECT:

```
SELECT * FROM V1
```

is dependent on view *V1*, table *T1,* and external scalar function *SIN.*

DROP Statements

Use Drop statements with functions, indexes, procedures, schemas, synonyms, tables, triggers, and views.

DROP FUNCTION statement

Syntax

```
DROP FUNCTION function-name
```

Identifies the particular function to be dropped, and is valid only if there is exactly one function instance with the *function-name* in the schema. The identified function can have any number of parameters defined for it. If no function with the indicated name in the named or implied schema, an error (SQLSTATE 42704) will occur. An error will also occur if there is more than one specific instance of the function in the named or implied schema.

DROP INDEX statement

DROP INDEX removes the specified index.

Syntax

```
DROP INDEX
index-Name
```

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

Statement 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 PROCEDURE statement

Syntax

```
DROP PROCEDURE
procedure-Name
```

Identifies the particular procedure to be dropped, and is valid only if there is exactly one

procedure instance with the *procedure-name* in the schema. The identified procedure can have any number of parameters defined for it. If no procedure with the indicated name in the named or implied schema, an error (SQLSTATE 42704) will occur. An error will also occur if there is more than one specific instance of the procedure in the named or implied schema.

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.

Syntax

```
DROP SCHEMA
schemaName
 RESTRICT
```

The RESTRICT keyword enforces the rule that no objects can be defined in the specified schema for the schema to be deleted from the database. The RESTRICT keyword is required

```
-- Drop the SAMP schema
-- The SAMP schema may only be deleted from the database
-- if no objects are defined in the SAMP schema.
DROP SCHEMA SAMP RESTRICT
```

DROP SYNONYM statement

Drops the specified synonym from a table or view.

Syntax

```
DROP SYNONYM
synonym-Name
```

DROP TABLE statement

DROP TABLE removes the specified table.

Syntax

```
DROP TABLE
table-Name
```

Statement dependency system

Triggers, constraints (primary, unique, check and references from the table being dropped) and indexes on the table are silently dropped. The existence of an open cursor that references table being dropped cause the DROP TABLE statement to generate an error, and the table is not dropped.

Dropping a table invalidates statements that depend on the table. (Invalidating a statement causes it to be recompiled upon the next execution. See Interaction with the dependency system .)

DROP TRIGGER statement

DROP TRIGGER removes the specified trigger.

Syntax

```
DROP TRIGGER
TriggerName
```

```
DROP TRIGGER TRIG1
```

Statement 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
view-Name
```

```
DROP VIEW AnIdentifier
```

Statement 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.

RENAME statements

Use the Rename statements with indexes and tables.

RENAME INDEX statement

This statement allows you to rename an index in the current schema. Users cannot rename indexes in the *SYS* schema.

Syntax

```
RENAME INDEX index-Name TO new-index-Name
```

```
RENAME INDEX DESTINDEX TO ARRIVALINDEX
```

Statement dependency system

RENAME INDEX is not allowed if there are any open cursors that reference the index being renamed.

RENAME TABLE statement

RENAME TABLE allows you to rename an existing table in any schema (except the schema *SYS*).

Syntax

```
RENAME TABLE table-Name TO
new-Table-Name
```

If there is a view or foreign key that references the table, attempts to rename it will generate an error. In addition, if there are any check constraints or triggers on the table, attempts to rename it will also generate an error.

```
RENAME TABLE SAMP.EMP_ACT TO EMPLOYEE_ACT
```

Also see ALTER TABLE statement for more information.

Statement dependency system

If there is an index defined on the table, the table can be renamed.

RENAME TABLE is not allowed if there are any open cursors that reference the table being altered.

SET statements

Use the Set statements with schemas and to set the current isolation level.

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. If the schema doesn't exist an error is returned. See CREATE SCHEMA statement .

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 [CURRENT] SCHEMA [=]
{
schemaName
|
USER | ? | '<string-constant>' } | SET CURRENT SQLID [=]
{
```

```
schemaName
| USER | ? | '<string-constant>' }
```

schemaName is an identifier with a maximum length of 128. It is case insensitive unless enclosed in double quotes. (For example, SYS is equivalent to sYs, SYs, sys, etcetera.)

USER is the current user. If no current user is defined, the current schema defaults the *APP* schema. (If a user name was specified upon connection, the user's name is the default schema for the connection, if a schema with that name exists.)

? is a dynamic parameter specification that can be used in prepared statements. The SET SCHEMA statement can be prepared once and then executed with different schema values. The schema values are treated as string constants so they are case sensitive. For example, to designate the *APP* schema, use the string "APP" rather than "app".

```
-- the following are all equivalent and will work
-- assuming a schema called HOTEL
SET SCHEMA HOTEL
SET SCHEMA hotel
SET CURRENT SCHEMA hotel
SET CURRENT SQLID hotel
SET SCHEMA = hotel
SET CURRENT SCHEMA = hotel
SET CURRENT SQLID = hotel
SET SCHEMA "HOTEL" -- quoted identifier
SET SCHEMA 'HOTEL' -- quoted string
--This example produces an error because
 --lower case hotel won't be found
SET SCHEMA = 'hotel'
 --This example produces an error because SQLID is not
 --allowed without CURRENT
SET SQLID hotel
 -- This sets the schema to the current user id
SET CURRENT SCHEMA USER
// Here's an example of using set schema in an Java program
PreparedStatement ps = conn.PrepareStatement("set schema ?");
ps.setString(1,"HOTEL");
ps.executeUpdate();
... do some work
ps.setString(1,"APP");
ps.executeUpdate();

ps.setString(1,"app"); //error - string is case sensitive
// no app will be found
ps.setNull(1, Types.VARCHAR); //error - null is not allowed
```

SET CURRENT ISOLATION statement

The SET CURRENT ISOLATION LEVEL statement allows a user to change the isolation level for the user's connection. Valid levels are SERIALIZABLE, REPEATABLE READ, READ COMMITTED, and READ UNCOMMITTED.

Issuing this command commits the current transaction, which is consistent with the *java.sql.Connection.setTransactionLevel* method.

For information about isolation levels, see "Locking, Concurrency, and Isolation" in the *Derby Developer's Guide* .

Syntax

```
SET [ CURRENT ] ISOLATION [ = ]
{
UR | DIRTY READ | READ UNCOMMITTED
CS | READ COMMITTED | CURSOR STABILITY
RS |
RR | REPEATABLE READ | SERIALIZABLE
RESET
```

```
}
```

```
set isolation serializable;
```

CALL (PROCEDURE)

The CALL (PROCEDURE) statement is used to call procedures. A call to a procedure does not return any value.

Syntax

```
CALL
procedure-Name
 ( [ ? [, ?]* ] )
```

Example

```
CREATE PROCEDURE SALES.TOTAL_REVENUE(IN S_MONTH INTEGER,
IN S_YEAR INTEGER, OUT TOTAL DECIMAL(10,2))
PARAMETER STYLE JAVA READS SQL DATA LANGUAGE JAVA EXTERNAL NAME
'com.acme.sales.calculateRevenueByMonth';
CALL SALES.TOTAL_REVENUE(?,?,?);
```

CONSTRAINT clause

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).

- PRIMARY KEY

  Specifies the column that uniquely identifies a row in the table. The identified columns must be defined as NOT NULL.

  Note: If you attempt to add a primary key using ALTER TABLE and any of the columns included in the primary key contain null values, an error will be generated and the primary key will not be added. See ALTER TABLE statement for more information.

- UNIQUE

  Specifies that values in the column must be unique. NULL values are not 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. The identified columns must be defined as NOT NULL.

- 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.

Syntax

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 constraint, none of the columns included in the primary key can have NULL constraints; that is, they must not permit NULL values.

ALTER TABLE ADD PRIMARY KEY allows you to include existing columns in a primary key if they were first defined as NOT NULL. NULL values are not allowed. If the column(s) contain NULL values, the system will not add the primary key constraint. See ALTER TABLE statement for more information.

A table can have at most one PRIMARY KEY constraint, but can have multiple UNIQUE constraints.

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*.)

A foreign key constraint is satisfied if there is a matching value in the referenced unique or primary key column. 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, Derby 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, Derby 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), Derby 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 Derby throws a statement exception.

Derby 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. Derby has already created it for you. See Indexes and constraints .

These indexes are available to the optimizer for query optimization (see CREATE INDEX statement) 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.

Requirements for search condition

If a check constraint is specified as part of a column-definition, a column reference can only be made to the same column. Check constraints specified as part of a table definition can have column references identifying columns previously defined in the CREATE TABLE statement.

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)
- Subqueries
- User Functions (such as USER, SESSION_USER, CURRENT_USER)

Referential actions

You can specify an ON DELETE clause and/or an ON UPDATE clause, followed by the appropriate action (CASCADE, RESTRICT, SET NULL, or NO ACTION) when defining foreign keys. These clauses specify whether Derby should modify corresponding foreign key values or disallow the operation, to keep foreign key relationships intact when a primary key value is updated or deleted from a table.

You specify the update and delete rule of a referential constraint when you define the referential constraint.

The update rule applies when a row of either the parent or dependent table is updated. The choices are NO ACTION and RESTRICT.

When a value in a column of the parent table's primary key is updated and the update rule has been specified as RESTRICT, Derby checks dependent tables for foreign key constraints. If any row in a dependent table violates a foreign key constraint, the transaction is rolled back.

If the update rule is NO ACTION, Derby checks the dependent tables for foreign key constraints *after* all deletes have been executed but *before* triggers have been executed. If any row in a dependent table violates a foreign key constraint, the statement is rejected.

When a value in a column of the dependent table is updated, and that value is part of a foreign key, NO ACTION is the implicit update rule. NO ACTION means that if a foreign key is updated with a non-null value, the update value must match a value in the parent table's primary key when the update statement is completed. If the update does not match a value in the parent table's primary key, the statement is rejected.

The delete rule applies when a row of the parent table is deleted and that row has dependents in the dependent table of the referential constraint. If rows of the dependent table are deleted, the delete operation on the parent table is said to be *propagated* to the dependent table. If the dependent table is also a parent table, the action specified applies, in turn, to its dependents.

The choices are NO ACTION, RESTRICT, CASCADE, or SET NULL. SET NULL can be specified only if some column of the foreign key allows null values.

If the delete rule is:

NO ACTION, Derby checks the dependent tables for foreign key constraints *after* all deletes have been executed but *before* triggers have been executed. If any row in a dependent table violates a foreign key constraint, the statement is rejected.

RESTRICT, Derby checks dependent tables for foreign key constraints. If any row in a

dependent table violates a foreign key constraint, the transaction is rolled back.

CASCADE, the delete operation is propagated to the dependent table (and that table's dependents, if applicable).

SET NULL, each nullable column of the dependent table's foreign key is set to null. (Again, if the dependent table also has dependent tables, nullable columns in those tables' foreign keys are also set to null.)

Each referential constraint in which a table is a parent has its own delete rule; all applicable delete rules are used to determine the result of a delete operation. Thus, a row cannot be deleted if it has dependents in a referential constraint with a delete rule of RESTRICT or NO ACTION. Similarly, a row cannot be deleted if the deletion cascades to any of its descendants that are dependents in a referential constraint with the delete rule of RESTRICT or NO ACTION.

Deleting a row from the parent table involves other tables. Any table involved in a delete operation on the parent table is said to be delete-connected to the parent table. The delete can affect rows of these tables in the following ways:

- If the delete rule is RESTRICT or NO ACTION, a dependent table is involved in the operation but is not affected by the operation. (That is, Derby checks the values within the table, but does not delete any values.)
- If the delete rule is SET NULL, a dependent table's rows can be updated when a row of the parent table is the object of a delete or propagated delete operation.
- If the delete rule is CASCADE, a dependent table's rows can be deleted when a parent table is the object of a delete.
- If the dependent table is also a parent table, the actions described in this list apply, in turn, to its dependents.

Examples

```
-- column-level primary key constraint named OUT_TRAY_PK:
CREATE TABLE SAMP.OUT_TRAY
        (
        SENT TIMESTAMP,
        DESTINATION CHAR(8),
        SUBJECT CHAR(64) NOT NULL CONSTRAINT OUT_TRAY_PK PRIMARY KEY,
        NOTE_TEXT VARCHAR(3000)
   );

-- the table-level primary key definition allows you to
-- include two columns in the primary key definition:
CREATE TABLE SAMP.SCHED
        (
        CLASS_CODE CHAR(7) NOT NULL,
        DAY SMALLINT NOT NULL,
        STARTING TIME,
        ENDING TIME,
        PRIMARY KEY (CLASS_CODE, DAY)
        );

-- Use a column-level constraint for an arithmetic check
-- Use a table-level constraint
-- to make sure that a employee's taxes does not
-- exceed the bonus
CREATE TABLE SAMP.EMP
        (
        EMPNO CHAR(6) NOT NULL CONSTRAINT EMP_PK PRIMARY KEY,
        FIRSTNME CHAR(12) NOT NULL,
        MIDINIT vARCHAR(12) NOT NULL,
        LASTNAME VARCHAR(15) NOT NULL,
        SALARY DECIMAL(9,2) CONSTRAINT SAL_CK CHECK (SALARY >= 10000),
        BONUS DECIMAL(9,2),
        TAX DECIMAL(9,2),
        CONSTRAINT BONUS_CK CHECK (BONUS > TAX)
        );

-- use a check constraint to allow only appropriate
-- abbreviations for the meals
CREATE TABLE FLIGHTS
        (
        FLIGHT_ID CHAR(6) NOT NULL ,
```

```
        SEGMENT_NUMBER INTEGER NOT NULL ,
        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')),
        PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER)
        );

CREATE TABLE METROPOLITAN
        (
        HOTEL_ID INT NOT NULL CONSTRAINT HOTELS_PK PRIMARY KEY,
        HOTEL_NAME VARCHAR(40) NOT NULL,
        CITY_ID INT CONSTRAINT METRO_FK REFERENCES CITIES
        );

-- create a table with a table-level primary key constraint
-- and a table-level foreign key constraint
CREATE TABLE FLTAVAIL
        (
        FLIGHT_ID CHAR(6) NOT NULL,
        SEGMENT_NUMBER INT NOT NULL,
        FLIGHT_DATE DATE NOT NULL,
        ECONOMY_SEATS_TAKEN INT,
        BUSINESS_SEATS_TAKEN INT,
        FIRSTCLASS_SEATS_TAKEN INT,
        CONSTRAINT FLTAVAIL_PK PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER),
        CONSTRAINT FLTS_FK
        FOREIGN KEY (FLIGHT_ID, SEGMENT_NUMBER)
        REFERENCES Flights (FLIGHT_ID, SEGMENT_NUMBER)
        );
-- add a unique constraint to a column
ALTER TABLE SAMP.PROJECT
ADD CONSTRAINT P_UC UNIQUE (PROJNAME);

-- create a table whose city_id column references the
-- primary key in the Cities table
-- using a column-level foreign key constraint
CREATE TABLE CONDOS
        (
        CONDO_ID INT NOT NULL CONSTRAINT hotels_PK PRIMARY KEY,
        CONDO_NAME VARCHAR(40) NOT NULL,
        CITY_ID INT CONSTRAINT city_foreign_key
        REFERENCES Cities ON DELETE CASCADE ON UPDATE RESTRICT
        );
```

Statement 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.

Column-level-constraint

```
{
    NOT NULL |
    [ [CONSTRAINT
constraint-Name
]
    {
        CHECK (
searchCondition
) |
        {
            PRIMARY KEY |
            UNIQUE |

REFERENCES clause

        }
    }
}
```

Table-level constraint

```
[CONSTRAINT
constraint-Name
]
{
    CHECK (
searchCondition
) |
    {
        PRIMARY KEY (
Simple-column-Name
 [ ,
Simple-column-Name
 ]* ) |
        UNIQUE (
Simple-column-Name
 [ ,
Simple-column-Name
 ]* ) |
        FOREIGN KEY (
Simple-column-Name
 [ ,
Simple-column-Name
 ]* )

REFERENCES clause

    }
}
```

References specification

```
REFERENCES
table-Name
 [ (
Simple-column-Name
 [ ,
Simple-column-Name
 ]* ) ]
[ ON DELETE {NO ACTION | RESTRICT | CASCADE | SET NULL}]
   [ ON UPDATE {NO ACTION | RESTRICT }]
|
[ ON UPDATE {NO ACTION | RESTRICT }] [ ON DELETE
   {NO ACTION | RESTRICT | CASCADE | SET NULL}]
```

searchCondition

A *searchCondition* is any Boolean expression that meets the requirements specified in Requirements for search condition .

If a *constraint-Name* is not specified, Derby generates a unique constraint name (for either column or table constraints).

DECLARE GLOBAL TEMPORARY TABLE statement

The DECLARE GLOBAL TEMPORARY TABLE statement defines a temporary table for the current connection. These tables do not reside in the system catalogs and are not persistent. Temporary tables exist only during the connection that declared them and cannot be referenced outside of that connection. When the connection closes, the rows of the table are deleted, and the in-memory description of the temporary table is dropped.

Temporary tables are useful when:

- the table structure is not known before using an application.
- other users do not need the same table structure.
- data in the temporary table is needed while using the application.
- the table can be declared and dropped without holding the locks on the system catalog.

Syntax

```
DECLARE GLOBAL TEMPORARY TABLE
table-Name

    { column-definition [ , column-definition ] * }
[ ON COMMIT {DELETE | PRESERVE} ROWS ]
NOT LOGGED [ON ROLLBACK DELETE ROWS]
```

table-Name

Names the temporary table. If a schema-Name other than SESSION is specified, an error will occur (SQLSTATE 428EK). If the schema-Name is not specified, SESSION is assigned. Multiple connections can define declared global temporary tables with the same name because each connection has its own unique table descriptor for it.

Using SESSION as the schema name of a physical table will not cause an error, but is discouraged. The SESSION schema name should be reserved for the temporary table schema.

column-definition

See column-definition for `CREATE TABLE` for more information on `column-definition`. `DECLARE GLOBAL TEMPORARY TABLE` does not allow `generated-column-spec` in the `column-definition`.

Data-type

Supported data-types are:

- BIGINT
- CHAR
- DATE
- DECIMAL
- DOUBLE PRECISION
- FLOAT
- INTEGER
- NUMERIC
- REAL
- SMALLINT
- TIME
- TIMESTAMP
- VARCHAR

ON COMMIT

Specifies the action taken on the global temporary table when a COMMIT operation is

performed.

DELETE ROWS

All rows of the table will be deleted if no hold-able cursor is open on the table. This is the default value for ON COMMIT. If you specify ON ROLLBACK DELETE ROWS, this will delete all the rows in the table only if the temporary table was used. ON COMMIT DELETE ROWS will delete the rows in the table even if the table was not used (if the table does not have hold-able cursors open on it).

PRESERVE ROWS

The rows of the table will be preserved.

NOT LOGGED

Specifies the action taken on the global temporary table when a rollback operation is performed. When a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed, if the table was created in the unit of work (or savepoint), the table will be dropped. If the table was dropped in the unit of work (or savepoint), the table will be restored with no rows.

ON ROLLBACK DELETE ROWS

This is the default value for NOT LOGGED. NOT LOGGED [ON ROLLBACK DELETE ROWS]] specifies the action that is to be taken on the global temporary table when a ROLLBACK or (ROLLBACK TO SAVEPOINT) operation is performed. If the table data has been changed, all the rows will be deleted.

Examples

```
set schema myapp;

create table t1(c11 int, c12 date);

declare global temporary table SESSION.t1(c11 int) not logged;
-- The SESSION qualification is redundant here because temporary
-- tables can only exist in the SESSION schema.

declare global temporary table t2(c21 int) not logged;
-- The temporary table is not qualified here with SESSION because
temporary
-- tables can only exist in the SESSION schema.

insert into SESSION.t1 values (1);
-- SESSION qualification is mandatory here if you want to use
-- the temporary table, because the current schema is "myapp."

select * from t1;
-- This select statement is referencing the "myapp.t1" physical
-- table since the table was not qualified by SESSION.
```

Note that temporary tables can only be declared in the SESSION schema. You should never declare a physical schema with the SESSION name.

The following is a list of DB2 UDB DECLARE GLOBAL TEMPORARY TABLE functions that are not supported by Derby:

- IDENTITY column-options
- IDENTITY attribute in copy-options
- AS (fullselect) DEFINITION ONLY
- NOT LOGGED ON ROLLBACK PRESERVE ROWS
- IN tablespace-name
- PARTITIONING KEY
- WITH REPLACE

Restrictions on Declared Global Temporary Tables

Temporary tables cannot be specified in the following statements:

- ALTER TABLE
- CREATE SYNONYM
- CREATE TRIGGER
- CREATE VIEW
- LOCK
- RENAME

Temporary tables cannot be specified in referential constraints.

There is no check constraints support for columns.

The following data types cannot be used with Declared Global Temporary Tables:

- BLOB
- CLOB
- LONG VARCHAR

Temporary tables cannot be referenced in a triggered-SQL-statement.

If a statement performing an insert, update, or delete to the temporary table encounters an error, all the rows of the table are deleted.

Restrictions Specific to Derby

Derby does not support the following on temporary tables:

- index support
- synonyms, triggers and views on SESSION schema tables (including physical tables and temporary tables)
- LOCK TABLE
- constraints and primary keys
- generated-column-spec
- importing into temporary tables

Any statements referencing SESSION schema tables and views will not be cached.

DELETE statement

Syntax

```
{
    DELETE FROM table-Name
        [
WHERE clause
] |
    DELETE FROM table-Name
WHERE CURRENT OF
}
```

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 .

Examples

```
DELETE FROM SAMP.IN_TRAY;

stmt.executeUpdate("DELETE FROM SAMP.IN_TRAY WHERE CURRENT OF " +
        resultSet.getCursorName())
```

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 has not 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.

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 .

Syntax

```
FOR
{
    READ ONLY | FETCH ONLY |
    UPDATE [ OF
Simple-column-Name
 [ ,
Simple-column-Name
]* ]
}
```

Simple-column-Name 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.

The optimizer is able to use an index even if the column in the index is being updated. For more information about how indexes affect cursors, see *Tuning Derby* .

```
SELECT RECEIVED, SOURCE, SUBJECT, NOTE_TEXT FROM SAMP.IN_TRAY FOR UPDATE;
```

FROM clause

The FROM clause is a mandatory clause in a *SelectExpression* . It specifies the tables (TableExpression) from which the other clauses of the query can access columns for use

in expressions.

Syntax

```
FROM TableExpression [ , TableExpression ] *
```

```
SELECT Cities.city_id
FROM Cities
WHERE city_id < 5
-- other types of TableExpressions
SELECT TABLENAME, ISINDEX
FROM SYS.SYSTABLES T, SYS.SYSCONGLOMERATES C
WHERE T.TABLEID = C.TABLEID
ORDER BY TABLENAME, ISINDEX;
-- 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 COUNTRIES.COUNTRY, CITIES.CITY_NAME, FLIGHTS.DEST_AIRPORT
FROM COUNTRIES LEFT OUTER JOIN CITIES
ON COUNTRIES.COUNTRY_ISO_CODE = CITIES.COUNTRY_ISO_CODE
LEFT OUTER JOIN FLIGHTS
ON Cities.AIRPORT = FLIGHTS.DEST_AIRPORT;
```

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
column-Name
 [ ,
column-Name
 ] *
```

column-Name 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.

```
-- find the average flying_times 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 a smallint
SELECT ID, AVG(SALARY)
FROM SAMP.STAFF
GROUP BY ID
-- Get the AVGSALARY and EMPCOUNT columns, and the DEPTNO column using
the AS clause
-- And group by the WORKDEPT column using the correlation name OTHERS
SELECT OTHERS.WORKDEPT AS DEPTNO,
AVG(OTHERS.SALARY) AS AVGSALARY,
COUNT(*) AS EMPCOUNT
FROM SAMP.EMPLOYEE OTHERS
GROUP BY OTHERS.WORKDEPT;
```

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. The SELECT clause cannot refer directly to any column that does not have a GROUP BY clause. It can, however, refer to constants, aggregates, and special registers.

Syntax

```
HAVING searchCondition
```

The *searchCondition*, which is a specialized *booleanExpression*, can contain only grouping columns (see GROUP BY clause), 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 SALARY is not a grouping column, it does not appear within an aggregate, and it is not within a subquery:

```
-- SELECT COUNT(*)
-- FROM SAMP.STAFF
-- GROUP BY ID
-- HAVING SALARY > 15000;
```

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.

```
-- 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 SUBSTR(FLIGHTAVAILABILITY.FLIGHT_ID, 1, 2) = AIRLINE
GROUP BY AIRLINE_FULL
HAVING COUNT(*) > 1
```

INNER JOIN

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

Syntax

```
TableExpression
 [ INNER ] JOIN
TableExpression
 { ON booleanExpression }
```

You can specify the join clause by specifying ON with a boolean expression.

The scope of expressions in the ON 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 SAMP.EMPLOYEE INNER JOIN SAMP.STAFF
ON EMPLOYEE.SALARY < STAFF.SALARY;
```

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).

```
-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result
SELECT SAMP.EMP_ACT.*, LASTNAME
     FROM SAMP.EMP_ACT JOIN SAMP.EMPLOYEE
     ON EMP_ACT.EMPNO = EMPLOYEE.EMPNO;
-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the
-- DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930.
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
     FROM SAMP.EMPLOYEE JOIN SAMP.DEPARTMENT
     ON WORKDEPT = DEPTNO
     AND YEAR(BIRTHDATE) < 1930;

-- Another example of "generating" new data values,
-- using a query which selects from a VALUES clause (which is an
-- alternate form of a fullselect).
-- This query shows how a table can be derived called "X"
-- having 2 columns "R1" and "R2" and 1 row of data
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;

This results in:
  C1         |C2          |C1          |2
-----------------------------------------------
3           |4           |3           |2
1           |5           |1           |2

-- List every department with the employee number and
-- last name of the manager

SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
        FROM DEPARTMENT INNER JOIN EMPLOYEE
        ON MGRNO = EMPNO;

-- List every employee number and last name
-- with the employee number and last name of their manager
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
        FROM EMPLOYEE E INNER JOIN
        DEPARTMENT INNER JOIN EMPLOYEE M
        ON MGRNO = M.EMPNO
        ON E.WORKDEPT = DEPTNO;
```

INSERT statement

An INSERT statement creates a row or rows and stores them in the named table. The number of values assigned in an INSERT statement must be the same as the number of specified or implied columns.

Syntax

```
INSERT INTO table-Name
    [ (
Simple-column-Name
 [ ,
Simple-column-Name
]* ) ]
          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 .

- UNION expressions

For more information about Query, see Query .

```
INSERT INTO COUNTRIES
      VALUES ('Taiwan', 'TW', 'Asia');

-- Insert a new department into the DEPARTMENT table,
-- but do not assign a manager to the new department
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
       VALUES ('E31', 'ARCHITECTURE', 'E01');
-- Insert two new departments using one statement
-- into the DEPARTMENT table as in the previous example,
-- but do not assign a manager to the new department.
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
       VALUES ('B11', 'PURCHASING', 'B01'),
       ('E41', 'DATABASE ADMINISTRATION', 'E01');
--  Create a temporary table MA_EMP_ACT with the
-- same columns as the EMP_ACT table.
-- Load MA_EMP_ACT with the rows from the EMP_ACT
-- table with a project number (PROJNO)
-- starting with the letters 'MA'.
CREATE TABLE MA_EMP_ACT
    (
        EMPNO CHAR(6)  NOT NULL,
        PROJNO CHAR(6)  NOT NULL,
        ACTNO SMALLINT  NOT NULL,
        EMPTIME DEC(5,2),
        EMSTDATE DATE,
        EMENDATE  DATE
    );

INSERT INTO MA_EMP_ACT
       SELECT * FROM EMP_ACT
       WHERE SUBSTR(PROJNO, 1, 2) = 'MA';
-- Insert the DEFAULT value for the LOCATION column
INSERT INTO DEPARTMENT
      VALUES ('E31', 'ARCHITECTURE', '00390', 'E01', DEFAULT);
```

Statement dependency system

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.

JOIN operation

The JOIN operations, which are among the possible *TableExpression s* 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 "WHERE t1.col1 = t2.col2".)

Syntax

```
JOIN Operation
```

The JOIN operations are:

- INNER JOIN

  Specifies a join between two tables with an explicit join clause. See INNER JOIN .

- 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 .

- 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 .

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

JOIN expressions and query optimization

For information on which types of joins are optimized, see *Tuning Derby* .

LEFT OUTER JOIN

A LEFT OUTER JOIN is one of the JOIN operations s 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
    }
```

The scope of expressions in either the ON 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).

```
--match cities to countries
SELECT CITIES.COUNTRY, REGION
FROM Countries
LEFT OUTER JOIN Cities
ON CITY_ID=CITY_ID
WHERE REGION = 'Asia';
-- use the synonymous syntax, RIGHT JOIN, to achieve exactly
-- the same results as in the example above

SELECT COUNTRIES.COUNTRY, REGION
FROM Countries
LEFT JOIN Cities
ON CITY_ID=CITY_ID;
-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table
-- and DEPTNO in the DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
   FROM SAMP.EMPLOYEE LEFT OUTER JOIN SAMP.DEPARTMENT
   ON WORKDEPT = DEPTNO
   AND YEAR(BIRTHDATE) < 1930;
-- List every department with the employee number and
-- last name of the manager,
-- including departments without a manager
SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
      FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
            ON MGRNO = EMPNO;
```

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
table-Name
 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. Replace line 13 with this: For example, if a transaction locks the entire Flights table in share mode in order to read data, a particular statement might 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.

```
-- 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 {
column-Name
 | ColumnPosition }
    [ ASC | DESC ]
    [ ,
column-Name
 | ColumnPosition
        [ ASC | DESC ] ] *
```

ColumnPosition is an integer that identifies the number of the column in the SelectItem in the underlying Query of the SELECT statement . *ColumnPosition* must be greater than 0 and not greater than the number of columns in the result table. In other words, if you want to order by a column, that column must be in the select list.

column-Name refers to the names visible from the *SelectItems* in the underlying query of the SELECT statement . An order by column does not need to 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 and updatable ResultSets .)

For example, if an INTEGER column contains integers, NULL is considered greater than

1 for purposes of sorting. In other words, NULL values are sorted high.

```
-- order by the correlation name NATION
-- order by the correlation name NATION
SELECT CITY_NAME, COUNTRY AS NATION
FROM CITIES
ORDER BY NATION;
```

Query

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

Syntax

```
{
    ( Query ) |
    Query INTERSECT [ ALL | DISTINCT ] Query |
    Query EXCEPT [ ALL | DISTINCT ] Query |
    Query UNION [ ALL | DISTINCT ] Query |

SelectExpression

 |

VALUES Expression

}
```

You can arbitrarily put parentheses around queries, or use the parentheses to control the order of evaluation of the INTERSECT, EXCEPT, or UNION operations. These operations are evaluated from left to right when no parentheses are present, with the exception of INTERSECT operations, which would be evaluated before any UNION or EXCEPT operations.

Duplicates in UNION, INTERSECT, and EXCEPT ALL results

The ALL and DISTINCT keywords determine whether duplicates are eliminated from the result of the operation. If you specify the DISTINCT keyword, then the result will have no duplicate rows. If you specify the ALL keyword, then there may be duplicates in the result, depending on whether there were duplicates in the input. DISTINCT is the default, so if you don't specify ALL or DISTINCT, the duplicates will be eliminated. For example, 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.

Depending on which operation is specified, if the number of copies of a row in the left table is L and the number of copies of that row in the right table is R, then the number of duplicates of that particular row that the output table contains (assuming the ALL keyword is specified) is:

- UNION: (L + R).
- EXCEPT: the maximum of (L – R) and 0 (zero).
- INTERSECT: the minimum of L and R.

Examples

```
-- a Select expression
SELECT *
FROM ORG;

-- a subquery
```

```
SELECT *
FROM (SELECT CLASS_CODE FROM CL_SCHED) AS CS;

-- a subquery
SELECT *
FROM (SELECT CLASS_CODE FROM CL_SCHED) AS CS (CLASS_CODE);

-- a UNION
-- returns all rows from columns DEPTNUMB and MANAGER
-- in table ORG
-- and (1,2) and (3,4)
-- DEPTNUMB and MANAGER are smallint columns
SELECT DEPTNUMB, MANAGER
FROM ORG
UNION ALL
VALUES (1,2), (3,4);

-- a values expression
VALUES (1,2,3);
-- List the employee numbers (EMPNO) of all employees in the EMPLOYEE
table
-- whose department number (WORKDEPT) either begins with 'E' or
-- who are assigned to projects in the EMP_ACT table
-- whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'
SELECT EMPNO
     FROM EMPLOYEE
     WHERE WORKDEPT LIKE 'E%'
  UNION
  SELECT EMPNO
     FROM EMP_ACT
     WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112');
-- Make the same query as in the previous example
-- and "tag" the rows from the EMPLOYEE table with 'emp' and
-- the rows from the EMP_ACT table with 'emp_act'.
-- Unlike the result from the previous example,
-- this query may return the same EMPNO more than once,
-- identifying which table it came from by the associated "tag"
SELECT EMPNO, 'emp'
     FROM EMPLOYEE
     WHERE WORKDEPT LIKE 'E%'
  UNION
  SELECT EMPNO, 'emp_act' FROM EMP_ACT
     WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112');
-- Make the same query as in the previous example,
-- only use UNION ALL so that no duplicate rows are eliminated
SELECT EMPNO
      FROM EMPLOYEE
      WHERE WORKDEPT LIKE 'E%'
  UNION ALL
  SELECT EMPNO
     FROM EMP_ACT
     WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112');
-- Make the same query as in the previous example,
-- only include an additional two employees currently not in any table
and
-- tag these rows as "new"
 SELECT EMPNO, 'emp'
     FROM EMPLOYEE
     WHERE WORKDEPT LIKE 'E%'
  UNION
  SELECT EMPNO, 'emp_act'
     FROM EMP_ACT
     WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
  UNION
     VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new');
```

RIGHT OUTER JOIN

A RIGHT OUTER JOIN is one of the JOIN operations s 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
    }
```

The scope of expressions in the ON 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).

```
-- get all countries and corresponding cities, including
-- countries without any cities
SELECT CITY_NAME, CITIES.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_NAME, CITIES.COUNTRY
FROM CITIES RIGHT OUTER JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE;
WHERE Countries.region = 'frica';
-- use the synonymous syntax, RIGHT JOIN, to achieve exactly
-- the same results as in the example above
SELECT CITY_NAME, CITIES.COUNTRY
FROM CITIES RIGHT JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE Countries.region = 'Africa';
-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
-- List every employee number and last name
-- with the employee number and last name of their manager
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
        FROM EMPLOYEE E RIGHT OUTER JOIN
        DEPARTMENT RIGHT OUTER JOIN EMPLOYEE M
        ON MGRNO = M.EMPNO
        ON E.WORKDEPT = DEPTNO;
```

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
)
```

```
-- avg always returns a single value, so the subquery is
-- a ScalarSubquery
SELECT NAME, COMM
  FROM STAFF
  WHERE EXISTS
    (SELECT AVG(BONUS + 800)
       FROM EMPLOYEE
       WHERE COMM < 5000
       AND EMPLOYEE.LASTNAME = UPPER(STAFF.NAME)
```

```
);
-- Introduce a way of "generating" new data values,
-- using a query which selects from a VALUES clause (which is an
alternate form of a fullselect).
-- This query shows how a table can be derived called "X" having
2 columns "R1" and "R2" and 1 row of data.
SELECT R1,R2
       FROM (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2);
```

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:

```
{
    * |
    {
table-Name
 |
correlation-Name
 } .* |
    Expression [AS
Simple-column-Name
]
}
```

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:

```
-- List head count of each department,
-- the department number (WORKDEPT), and the average departmental salary
(SALARY)
-- for all departments in the EMPLOYEE table.
-- Arrange the result table in ascending order by average departmental
salary.
SELECT WORKDEPT, AVG(SALARY)
     FROM EMPLOYEE
     GROUP BY WORKDEPT
     ORDER BY 1;
```

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.

Derby 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 .

The * wildcard

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

table-Name .* and *correlation-Name* .* 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, INTERSECT, or EXCEPT operator, the names from the first *SelectExpression* are taken as the names for the columns in the result of the operation. 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.

```
-- this example shows SELECT-FROM-WHERE
-- with an ORDER BY clause
-- and correlation-Names 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
ORDER BY CONSTRAINTNAME;
-- This example shows the use of the DISTINCT clause
SELECT DISTINCT ACTNO
FROM EMP_ACT;
-- This example shows how to rename an expression
-- Using the EMPLOYEE table, list the department number (WORKDEPT) and
-- maximum departmental salary (SALARY) renamed as BOSS
-- for all departments whose maximum salary is less than the
-- average salary in all other departments.
SELECT WORKDEPT AS DPT, MAX(SALARY) AS BOSS
        FROM EMPLOYEE EMP_COR
        GROUP BY WORKDEPT
        HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                                FROM EMPLOYEE
                                WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
        ORDER BY BOSS;
```

SELECT statement

A SELECT statement consists of a query with an optional ORDER BY clause and an optional FOR UPDATE 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, INTERSECT, and EXCEPT 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 SELECT statement supports the FOR FETCH ONLY clause. The FOR FETCH ONLY clause is synonymous with the FOR READ ONLY clause.

Remember: In order to get an updatable ResultSet, you must include a FOR UPDATE clause with the SELECT clause.

Syntax

```
Query

[
ORDER BY clause
]
[
FOR UPDATE clause
]
WITH {RR|RS|CS|UR}
```

You can set the isolation level in a SELECT statement using the WITH {RR|RS|CS|UR} syntax.

```
-- lists the names of the expression SAL+BONUS+COMM as TOTAL_PAY and
-- orders by the new name TOTAL_PAY
 SELECT FIRSTNME, SALARY+BONUS+COMM AS TOTAL_PAY
     FROM EMPLOYEE
     ORDER BY TOTAL_PAY;
-- creating an updatable cursor with a FOR UPDATE clause
-- to update the start date (PRSTDATE) and the end date (PRENDATE)
-- columns in the PROJECT table
SELECT PROJNO, PRSTDATE, PRENDATE
      FROM PROJECT
      FOR UPDATE OF PRSTDATE, PRENDATE;
-- set the isolation level to RR for this statement only
SELECT *
FROM Flights
WHERE flight_id BETWEEN 'AA1111' AND 'AA1112'
WITH RR;
```

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.

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).

Requirements for updatable cursors and updatable ResultSets
Only simple, single-table SELECT cursors and FORWARD_ONLY ResultSets can be updatable. The SELECT statement for updatable ResultSets has the same syntax as the SELECT statement for updatable cursors. 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
  - *SelectExpression s*
  - subqueries

There is no SQL 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" in Chapter 5 of the *Derby 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).

Statement 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).

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.

TableExpression

A *TableExpression* specifies a table or view in a FROM clause . It is the source from which a *SelectExpression* selects a result.

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.

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.

The Query in a *TableSubquery* appearing in a *FromItem* can contain multiple columns and return multiple rows. See *TableSubquery* .

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

Syntax

```
{
TableOrViewExpression | JOIN operation
}
```

```
-- SELECT from a Join expression
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
 FROM EMPLOYEE E LEFT OUTER JOIN
           DEPARTMENT INNER JOIN EMPLOYEE M
  ON MGRNO = M.EMPNO
  ON E.WORKDEPT = DEPTNO
```

TableOrViewExpression

```
{
```

```
table-Name
 |
view-Name
}
    [ [ AS ]
correlation-Name

        [ (
Simple-column-Name
 [ ,
Simple-column-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.

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

```
(
Query
)
```

```
-- 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
    (SELECT flight_ID
    FROM Flights WHERE orig_airport = 'SFO'
    OR dest_airport = 'SCL')
-- a subquery used with a quantified comparison
SELECT NAME, COMM
```

```
FROM STAFF
WHERE COMM >
(SELECT AVG(BONUS + 800)
     FROM EMPLOYEE
     WHERE COMM < 5000);
```

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
table-Name

        SET
column-Name
 =
Value

        [ ,
column-Name
 =
Value
} ]*
        [
WHERE clause
] |
    UPDATE
table-Name

        SET
column-Name
 =
Value

        [ ,
column-Name
 =
Value
 ]*

WHERE CURRENT OF

}
```

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 cursor is read-only and cannot be used to update.

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

```
-- All the employees except the manager of department (WORKDEPT) 'E21'
have
been temporarily reassigned.
-- Indicate this by changing their job (JOB) to NULL and their pay
-- (SALARY, BONUS, COMM) values to zero in the EMPLOYEE table.
UPDATE EMPLOYEE
  SET JOB=NULL, SALARY=0, BONUS=0, COMM=0
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'

-- PROMOTE the job (JOB) of certain employees to MANAGER
UPDATE EMPLOYEE
        SET JOB = 'MANAGER'
        WHERE CURRENT OF CURS1;
-- Increase the project staffing (PRSTAFF) by 1.5 for all projects
stmt.executeUpdate("UPDATE PROJECT SET PRSTAFF = "
"PRSTAFF + 1.5" +
"WHERE CURRENT OF" + ResultSet.getCursorName());

-- Change the job (JOB) of employee number (EMPNO) '000290' in the
EMPLOYEE table
-- to its DEFAULT value which is NULL
UPDATE EMPLOYEE
  SET JOB = DEFAULT
  WHERE EMPNO = '000290';
```

Statement dependency system

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 has not 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.

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.

Value

```
Expression | DEFAULT
```

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.

Syntax

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

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.

```
-- 3 rows of 1 column
VALUES (1),(2),(3);
-- 3 rows of 1 column
VALUES 1, 2, 3;
-- 1 row of 3 columns
VALUES (1, 2, 3);
-- 3 rows of 2 columns
VALUES (1,21),(2,22),(3,23);
-- constructing a derived table
VALUES ('orange', 'orange'), ('apple', 'red'),
('banana', 'yellow')
-- Insert two new departments using one statement into the DEPARTMENT
table,
-- but do not assign a manager to the new department.
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
  VALUES ('B11', 'PURCHASING', 'B01'),
    ('E41', 'DATABASE ADMINISTRATION', 'E01')
-- insert a row with a DEFAULT value for the MAJPROJ column
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE,
MAJPROJ)
VALUES ('PL2101', 'ENSURE COMPAT PLAN', 'B01', '000020', 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 a value returned by a built-in function
values char(1)
```

Value

```
Expression
 | DEFAULT
```

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
```

Boolean expressions are allowed in the WHERE clause. Most of the general expressions listed in Table of Expressions , can result in a boolean value.

In addition, there are the more common boolean expressions. Specific boolean operators listed in Table 10, take one or more operands; the expressions return a boolean value.

```
-- 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
-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result.
SELECT SAMP.EMP_ACT.*, LASTNAME
  FROM SAMP.EMP_ACT, SAMP.EMPLOYEE
  WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO;
-- Determine the employee number and salary of sales representatives
-- along with the average salary and head count of their departments.
-- This query must first create a new-column-name specified in the AS
clause
-- which is outside the fullselect (DINFO)
-- in order to get the AVGSALARY and EMPCOUNT columns,
-- as well as the DEPTNO column that is used in the WHERE clause
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
 FROM EMPLOYEE THIS_EMP,
   (SELECT OTHERS.WORKDEPT AS DEPTNO,
           AVG(OTHERS.SALARY) AS AVGSALARY,
           COUNT(*) AS EMPCOUNT
    FROM EMPLOYEE OTHERS
    GROUP BY OTHERS.WORKDEPT
   )AS DINFO
 WHERE THIS_EMP.JOB = 'SALESREP'
           AND THIS_EMP.WORKDEPT = DINFO.DEPTNO;
```

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 .

Syntax

```
WHERE CURRENT OF
cursor-Name
```

```
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 keyword or special operator executes some operation. Built-in functions use keywords or special built-in operators. Built-ins are SQL92Identifiers and are case-insensitive. Note that escaped functions like TIMESTAMPADD and TIMESTAMPDIFF are only accessible using the JDBC escape function syntax, and can be found in JDBC escape syntax .

Standard built-in functions

- ABS or ABSVAL
- BIGINT
- CAST
- CHAR
- Concatenation
- NULLIF and CASE expressions
- CURRENT_DATE
- CURRENT ISOLATION
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- DATE
- DAY
- DOUBLE
- HOUR
- IDENTITY_VAL_LOCAL
- INTEGER
- LENGTH
- LOCATE
- LCASE or LOWER
- LTRIM
- MINUTE
- MOD
- MONTH
- RTRIM
- SECOND
- SESSION_USER

- SMALLINT
- SQRT
- SUBSTR
- TIME
- TIMESTAMP
- UCASE or UPPER
- USER
- VARCHAR
- YEAR

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.

The built-in aggregates can operate on the data types shown in Permitted Data Types for Built-in Aggregates .

Table1. Permitted Data Types for Built-in Aggregates

'	All Types	Numeric Built-in Data Types
COUNT	X	X
MIN	'	X
MAX	'	X
AVG	'	X
SUM	'	X

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* .

All expressions in *SelectItems* in the *SelectExpression* must be either aggregates or grouped columns (see GROUP BY clause). (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:

```
-- 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 valid because SUM operates on a column from the outer query:

```
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
- COUNT
- MAX
- MIN
- SUM

ABS or ABSVAL

ABS or ABSVAL returns the absolute value of a numeric expression. The return type is the type of parameter. All built-in numeric types are supported (DECIMAL , DOUBLE PRECISION , FLOAT , INTEGER , BIGINT , NUMERIC , REAL , and SMALLINT).

Syntax

```
ABS(NumericExpression)
```

```
-- returns 3
VALUES ABS(-3)
```

AVG

AVG is an aggregate function that evaluates the average of an expression over a set of rows (see Aggregates (set functions)). AVG is allowed only on expressions that evaluate to numeric data types.

Syntax

```
AVG ( [ DISTINCT | ALL ] Expression )
```

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 valid:

```
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 an SQL-92 numeric data type. You can therefore call methods that evaluate to SQL-92 data types. 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 INTEGER *1*, which might not be what you would expect:

```
SELECT AVG(c1)
FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1)
```

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

```
SELECT AVG(CAST (c1 AS DOUBLE PRECISION))
FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1)
```

BIGINT

The BIGINT function returns a 64-bit integer representation of a number or character string in the form of an integer constant.

Syntax

```
BIGINT (CharacterExpression | NumericExpression )
```

CharacterExpression
An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The character string cannot be a long string. If the argument is a CharacterExpression, the result is the same number that would occur if the corresponding integer constant were assigned to a big integer column or variable.

NumericExpression
An expression that returns a value of any built-in numeric data type. If the argument is a NumericExpression, the result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

The result of the function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Using the EMPLOYEE table, select the EMPNO column in big integer form for further processing in the application:

```
SELECT BIGINT (EMPNO) FROM EMPLOYEE
```

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)
```

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*.

CAST conversions among SQL-92 data types
The following table shows valid explicit conversions between source types and target types for SQL data types.
Table1. Explicit conversions between source types and target types for SQL data types
This table shows which explicit conversions between data types are valid. The first column on the table lists the source types, while the first row lists the target types. A "Y" indicates that the source to the target is a valid conversion.

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
FLOAT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y
VARCHAR	Y	Y	Y	Y	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y
LONG VARCHAR	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-
CHAR FOR BIT	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP
DATA																		
VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-
LONG VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-
CLOB	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-
DATE	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	Y
TIME	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	Y
TIME STAMP	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	Y	Y

If a conversion is valid, CASTs are allowed. Size incompatibilities between the source and target types might cause runtime errors.

Notes

In this discussion, the Derby SQL-92 data types are categorized as follows:

- *numeric*
  - exact numeric (SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC)
  - approximate numeric (FLOAT, REAL, DOUBLE PRECISION)
- *string*
  - character string (CLOB, CHAR, VARCHAR, LONG VARCHAR)
  - bit string (BLOB, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, LONG VARCHAR FOR BIT DATA)
- *date/time*
  - DATE
  - TIME
  - TIMESTAMP

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.

Conversions from and to bit strings

Bit strings can be converted to other bit strings, but not character strings. Strings that are converted to bit strings are padded with trailing zeros to fit the size of the target bit string. The BLOB type is more limited and requires explicit casting. In most cases the BLOB type cannot be casted to and from other types.

Conversions of date/time values

A date/time value can always be converted to and from a TIMESTAMP. If a DATE is converted to a TIMESTAMP, the TIME component of the resulting TIMESTAMP is always 00:00:00. If a TIME data value is converted to a TIMESTAMP, the DATE component is set to the value of CURRENT_DATE at the time the CAST is executed. If a TIMESTAMP is converted to a DATE, the TIME component is silently truncated. If a TIMESTAMP is converted to a TIME, the DATE component is silently truncated.

```
SELECT CAST (miles AS INT)
FROM Flights
-- 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 a double as a decimal
SELECT CAST (FLYING_TIME AS DECIMAL(5,2))
FROM FLIGHTS
-- cast a SMALLINT to a BIGINT
VALUES CAST (CAST (12 as SMALLINT) as BIGINT)
```

CHAR

The CHAR function returns a fixed-length character string representation of:

- a character string, if the first argument is any type of character string.
- a datetime value, if the first argument is a date, time, or timestamp.
- a decimal number, if the first argument is a decimal number.
- a double-precision floating-point number, if the first argument is a DOUBLE or REAL.
- an integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT.

The first argument must be of a built-in data type. The result of the function is a fixed-length character string. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Character to character syntax

```
CHAR (CharacterExpression [, integer] )
```

CharacterExpression
An expression that returns a value that is CHAR, VARCHAR, LONG VARCHAR, or CLOB data type.

integer
The length attribute for the resulting fixed length character string. The value must be between 0 and 254.

If the length of the character-expression is less than the length attribute of the result, the result is padded with blanks up to the length of the result. If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning is returned unless the truncated characters were all blanks and the character-expression was not a long string (LONG VARCHAR or CLOB).

Integer to character syntax

```
CHAR (IntegerExpression )
```

IntegerExpression
An expression that returns a value that is an integer data type (either SMALLINT, INTEGER or BIGINT).

The result is the character string representation of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. It is left justified.

- If the first argument is a small integer: The length of the result is 6. If the number of characters in the result is less than 6, then the result is padded on the right with blanks to length 6.
- If the first argument is a large integer: The length of the result is 11. If the number of characters in the result is less than 11, then the result is padded on the right with blanks to length 11.
- If the first argument is a big integer: The length of the result is 20. If the number of characters in the result is less than 20, then the result is padded on the right with blanks to length 20.

Datetime to character syntax

```
CHAR (DatetimeExpression )
```

DatetimeExpression
An expression that is one of the following three data types:
- **date**: The result is the character representation of the date. The length of the result is 10.
- **time**: The result is the character representation of the time. The length of the result is 8.
- **timestamp**: The result is the character string representation of the timestamp. The length of the result is 26.

Decimal to character

```
CHAR (DecimalExpression )
```

DecimalExpression
An expression that returns a value that is a decimal data type. If a different precision and scale is desired, the DECIMAL scalar function can be used first to make the change.

Floating point to character syntax

```
CHAR (FloatingPointExpression )
```

FloatingPointExpression
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

Use the CHAR function to return the values for EDLEVEL (defined as smallint) as a fixed length character string:

```
SELECT CHAR(EDLEVEL) FROM EMPLOYEE
```

An EDLEVEL of 18 would be returned as the CHAR(6) value '18 ' (18 followed by four blanks).

LENGTH

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

```
LENGTH ( { CharacterExpression | BitExpression } )
```

```
-- returns 20
VALUES LENGTH('supercalifragilistic')
-- returns 1
VALUES LENGTH(X'FF')
-- returns 4
VALUES LENGTH(1234567890)
```

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 CHAR FOR BIT DATA, the resulting type is CHAR FOR BIT DATA; otherwise, it is VARCHAR FOR BIT DATA.

```
--returns 'supercalifragilisticexbealidocious(sp?)'
VALUES 'supercalifragilistic' || 'exbealidocious' || '(sp?)'
-- returns NULL
VALUES CAST (null AS VARCHAR(7))|| 'AString'
-- returns '130asdf'
VALUES '130' || 'asdf'
```

NULLIF and CASE expressions

Use the CASE and NULLIF expressions for conditional expressions in Derby.

NULLIF expression syntax

```
NULLIF(L, R)
```

The NULLIF expression is very similar to the CASE expression. For example:

```
NULLIF(V1,V2)
```

is equivalent to the following CASE expression:

```
CASE WHEN V1=V2 THEN NULL ELSE V1 END
```

CASE expression syntax

You can place a CASE expression anywhere an expression is allowed. It chooses an expression to evaluate based on a boolean test.

```
CASE WHEN BooleanExpression THEN thenExpression ELSE elseExpression END
```

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.

You do not need to use the CASE expression for avoiding *NullPointerExceptions* when a nullable column becomes a method receiver.

```
-- returns 3
VALUES CASE WHEN 1=1 THEN 3 ELSE 4 END;
```

If the value of the instance specified in an instance method invocation is null, the result of the invocation is null (SQL NULL). However, you still might need to use the CASE expression for when a nullable column is a primitive method parameter.

COUNT

COUNT is an aggregate function that counts the number of rows accessed in an expression (see Aggregates (set functions)). COUNT is allowed on all types of expressions.

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 of COUNT is BIGINT .

```
-- 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
```

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 BIGINT .

```
-- Count the number of rows in the Flights table
SELECT COUNT(*)
FROM Flights
```

CURRENT DATE

CURRENT DATE is a synonym for CURRENT_DATE .

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
```

or, alternately

```
CURRENT DATE
```

```
-- find available future flights:
SELECT * FROM Flightavailability where flight_date > CURRENT_DATE;
```

CURRENT ISOLATION

CURRENT ISOLATION returns the current isolation level as a char(2) value of either ""(blank), "UR", "CS", "RS", or "RR".

Syntax

```
CURRENT ISOLATION
```

```
VALUES CURRENT ISOLATION
```

CURRENT SCHEMA

CURRENT SCHEMA returns the schema name used to qualify unqualified database object references.

Note: CURRENT SCHEMA and CURRENT SQLID are synonyms.

These functions return a string of up to 128 characters.

Syntax

```
CURRENT SCHEMA
-- or, alternately:
CURRENT SQLID
```

```
-- Set the name column default to the current schema:
CREATE TABLE mytable (id int, name VARCHAR(128) DEFAULT CURRENT SQLID)
-- Inserts default value of current schema value into the table:
INSERT INTO mytable(id) VALUES (1)
-- Returns the rows with the same name as the current schema:
SELECT name FROM mytable WHERE name = CURRENT SCHEMA
```

CURRENT TIME

CURRENT TIME is a synonym for CURRENT_TIME .

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
```

or, alternately

```
CURRENT TIME
```

```
VALUES CURRENT_TIME
-- or, alternately:
VALUES CURRENT TIME
```

CURRENT TIMESTAMP

CURRENT TIMESTAMP is a synonym for CURRENT_TIMESTAMP .

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
```

or, alternately

```
CURRENT TIMESTAMP
```

```
VALUES CURRENT_TIMESTAMP
-- or, alternately:
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 and SESSION_USER are synonyms.

These functions return a string of up to 128 characters.

Syntax

```
CURRENT_USER
```

```
VALUES CURRENT_USER
```

DATE

The DATE function returns a date from a value. The argument must be a date, timestamp, a positive number less than or equal to 3,652,059, a valid string

representation of a date or timestamp, or a string of length 7 that is not a CLOB or LONG VARCHAR. If the argument is a string of length 7, it must represent a valid date in the form yyyynnn, where yyyy are digits denoting a year, and nnn are digits between 001 and 366, denoting a day of that year. The result of the function is a date. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp: The result is the date part of the value.
- If the argument is a number: The result is the date that is n-1 days after January 1, 0001, where n is the integral part of the number.
- If the argument is a string with a length of 7: The result is the date represented by the string.

Syntax

```
DATE ( expression )
```

This example results in an internal representation of '1988-12-25'.

```
VALUES DATE('1988-12-25')
```

DAY

The DAY function returns the day part of a value. The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp: The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a time duration or timestamp duration: The result is the day part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Syntax

```
DAY ( expression )
```

```
values day('2005-08-02');
```

The resulting value is 2.

DOUBLE

The DOUBLE function returns a floating-point number corresponding to a:

- number if the argument is a numeric expression.
- character string representation of a number if the argument is a string expression.

Numeric to double

```
DOUBLE [PRECISION] (NumericExpression )
```

NumericExpression
The argument is an expression that returns a value of any built-in numeric data type.

The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value. The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable.

Character string to double

```
DOUBLE (StringExpression )
```

StringExpression
The argument can be of type CHAR or VARCHAR in the form of a numeric constant. Leading and trailing blanks in argument are ignored.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value. The result is the same number that would occur if the string was considered a constant and assigned to a double-precision floating-point column or variable.

HOUR

The HOUR function returns the hour part of a value. The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp: The result is the hour part of the value, which is an integer between 0 and 24.
- If the argument is a time duration or timestamp duration: The result is the hour part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Syntax

```
HOUR ( expression )
```

Select all the classes that start in the afternoon from a table called TABLE1.

```
SELECT * FROM TABLE1
WHERE HOUR(STARTING) BETWEEN 12 AND 17
```

IDENTITY_VAL_LOCAL

Derby supports the IDENTITY_VAL_LOCAL function.

Syntax:

```
 IDENTITY_VAL_LOCAL ( )
```

The IDENTITY_VAL_LOCAL function is a non-deterministic function that returns the most recently assigned value of an identity column for a connection, where the assignment occurred as a result of a single row INSERT statement using a VALUES clause.

The IDENTITY_VAL_LOCAL function has no input parameters. The result is a DECIMAL (31,0), regardless of the actual data type of the corresponding identity column.

The value returned by the IDENTITY_VAL_LOCAL function, for a connection, is the value assigned to the identity column of the table identified in the most recent single row INSERT statement. The INSERT statement must contain a VALUES clause on a table containing an identity column. The assigned value is an identity value generated by Derby. The function returns a null value when a single row INSERT statement with a VALUES clause has not been issued for a table containing an identity column.

The result of the function is not affected by the following:

- A single row INSERT statement with a VALUES clause for a table without an identity column
- A multiple row INSERT statement with a VALUES clause
- An INSERT statement with a fullselect

If a table with an identity column has an INSERT trigger defined that inserts into another table with another identity column, then the IDENTITY_VAL_LOCAL() function will return the generated value for the statement table, and not for the table modified by the trigger.

Examples:

```
ij> create table t1(c1 int generated always as identity, c2 int);
0 rows inserted/updated/deleted
ij> insert into t1(c2) values (8);
1 row inserted/updated/deleted
ij> values IDENTITY_VAL_LOCAL();
1
------------------------------
1
1 row selected
ij> select IDENTITY_VAL_LOCAL()+1, IDENTITY_VAL_LOCAL()-1 from t1;
1                                  |2
-----------------------------------------------------------------------
2                                  |0
1 row selected
ij> insert into t1(c2) values (IDENTITY_VAL_LOCAL());
1 row inserted/updated/deleted
ij> select * from t1;
C1             |C2
------------------------------
1              |8
2              |1
2 rows selected
ij> values IDENTITY_VAL_LOCAL();
1
------------------------------
2
1 row selected
ij> insert into t1(c2) values (8), (9);
2 rows inserted/updated/deleted
ij> -- multi-values insert, return value of the function should not
change
values IDENTITY_VAL_LOCAL();
1
------------------------------
2
1 row selected
ij> select * from t1;
C1             |C2
------------------------------
1              |8
2              |1
3              |8
4              |9
4 rows selected
```

```
ij> insert into t1(c2) select c1 from t1;
4 rows inserted/updated/deleted
-- insert with sub-select, return value should not change
ij> values IDENTITY_VAL_LOCAL();
1
------------------------------
2
1 row selected
ij> select * from t1;
C1          |C2
------------------------------
1           |8
2           |1
3           |8
4           |9
5           |1
6           |2
7           |3
8           |4
8 rows selected
```

INTEGER

The INTEGER function returns an integer representation of a number, character string, date, or time in the form of an integer constant.

Syntax

```
INT[EGER] (NumericExpression | CharacterExpression )
```

NumericExpression
An expression that returns a value of any built-in numeric data type. If the argument is a numeric-expression, the result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

CharacterExpression
An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The character string cannot be a long string. If the argument is a character-expression, the result is the same number that would occur if the corresponding integer constant were assigned to a large integer column or variable.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and employee number (EMPNO). The list should be in descending order of the calculated value:

```
SELECT INTEGER (SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
FROM EMPLOYEE
ORDER BY 1 DESC
```

LOCATE

If a specified substring is found within a specified search string, LOCATE returns the index at which the substring is found within the search string. If the substring is not found, LOCATE returns 0.

Syntax

```
LOCATE(CharacterExpression, CharacterExpression [, StartPosition] )
```

The second CharacterExpression is the search string and is searched from the beginning, unless *startPosition* is specified, in which case the search begins from position there; the index starts with 1. It returns 0 if the string is not found.

The return type for LOCATE is an integer.

```
-- returns 2
VALUES LOCATE('love', 'clover')
```

LCASE or LOWER

LCASE or LOWER takes a character expression as a parameter and returns a string in which all alpha characters have been converted to lowercase.

Syntax

```
LCASE or LOWER ( CharacterExpression )
```

A *CharacterExpression* is a CHAR, VARCHAR, or LONG VARCHAR data type or any built-in type that is implicitly converted to a string (except a bit expression).

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.

```
-- returns 'asd1#w'
VALUES LOWER('aSD1#w')

SELECT LOWER(flight_id) FROM Flights
```

LTRIM

LTRIM removes blanks from the beginning of a character string expression.

Syntax

```
LTRIM(CharacterExpression)
```

A *CharacterExpression* is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string.

LTRIM returns NULL if *CharacterExpression* evaluates to null.

```
-- returns 'asdf  '
VALUES LTRIM(' asdf  ')
```

MAX

MAX is an aggregate function that evaluates the maximum of the expression over a set of values (see Aggregates (set functions)). MAX is allowed only on expressions that evaluate to built-in data types (including CHAR, VARCHAR, DATE, TIME, CHAR FOR BIT DATA, etc.).

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. 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 can 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).

```
-- find the latest date in the FlightAvailability table
SELECT MAX (flight_date) FROM FlightAvailability
-- find the longest flight originating from each airport,
-- but only when the longest flight is over 10 hours
SELECT MAX(flying_time), orig_airport
FROM Flights
GROUP BY orig_airport
HAVING MAX(flying_time) > 10
```

MIN

MIN is an aggregate expression that evaluates the minimum of an expression over a set of rows (see Aggregates (set functions)). MIN is allowed only on expressions that evaluate to built-in data types (including CHAR, VARCHAR, DATE, TIME, etc.).

Syntax

```
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), MIN (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. 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 can affect the result.

The resulting data type is the same as the expression on which it operates (it will never overflow).

```
-- NOT valid:
SELECT DISTINCT flying_time, MIN(DISTINCT miles) from Flights
-- valid:
SELECT COUNT(DISTINCT flying_time), MIN(DISTINCT miles) from Flights
-- find the earliest date:
SELECT MIN (flight_date) FROM FlightAvailability;
```

MINUTE

The MINUTE function returns the minute part of a value. The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp: The result is the minute part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration: The result is the minute part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Syntax

```
MINUTE ( expression )
```

```
-- Select all classes that do not end on a full hour:
SELECT * FROM table1 WHERE MINUTE(ending) < 60;
```

MOD

MOD returns the remainder (modulus) of argument 1 divided by argument 2. The result is negative only if argument 1 is negative.

Syntax

```
mod(integer_type, integer_type)
```

The result of the function is:

- SMALLINT if both arguments are SMALLINT.
- INTEGER if one argument is INTEGER and the other is INTEGER or SMALLINT.
- BIGINT if one integer is BIGINT and the other argument is BIGINT, INTEGER, or SMALLINT.

The result can be null; if any argument is null, the result is the null value.

MONTH

The MONTH function returns the month part of a value. The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp: The result is the month part of the value, which is an integer between 1 and 12.
- If the argument is a date duration or timestamp duration: The result is the month part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Syntax

```
MONTH ( expression )
```

Select all rows from the EMPLOYEE table for people who were born (BIRTHDATE) in DECEMBER.

```
SELECT * FROM EMPLOYEE
WHERE MONTH(BIRTHDATE) = 12
```

RTRIM

RTRIM removes blanks from the end of a character string expression.

Syntax

```
RTRIM(CharacterExpression)
```

A CharacterExpression is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string.

RTRIM returns NULL if *CharacterExpression* evaluates to null.

```
-- returns ' asdf'
VALUES RTRIM(' asdf  ')
```

```
-- returns 'asdf'
VALUES RTRIM('asdf  ')
```

SECOND

The SECOND function returns the seconds part of a value. The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp: The result is the seconds part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration: The result is the seconds part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Syntax

```
SECOND ( expression )
```

Assume that the column RECEIVED (timestamp) has an internal value equivalent to 1988-12-25-17.12.30.000000.

```
SECOND(RECEIVED)
```

Returns the value 30.

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
```

```
VALUES SESSION_USER
```

SMALLINT

The SMALLINT function returns a small integer representation of a number or character string in the form of a small integer constant.

Syntax

```
SMALLINT (NumericExpression | CharacterExpression )
```

NumericExpression
An expression that returns a value of any built-in numeric data type. If the argument is

a NumericExpression, the result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. The decimal part of the argument is truncated if present.

CharacterExpression

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. However, the value of the constant must be in the range of small integers. The character string cannot be a long string. If the argument is a CharacterExpression, the result is the same number that would occur if the corresponding integer constant were assigned to a small integer column or variable.

The result of the function is a small integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

SQRT

Returns the square root of a floating point number; only the built-in types REAL , FLOAT , and DOUBLE PRECISION are supported. The return type for SQRT is the type of the parameter.

Note: To execute SQRT on other data types, you must cast them to floating point types.

Syntax

```
SQRT(FloatingPointExpression)
```

```
-- throws an exception if any row stores a negative number:
VALUES SQRT(3421E+09)

-- returns the square root of an INTEGER after casting it as a
-- floating point data type:
SELECT SQRT(myDoubleColumn) FROM MyTable

VALUES SQRT (CAST(25 AS FLOAT));
```

SUBSTR

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 VARCHAR FOR BIT DATA in the second case. The length of the result is the maximum length of the source type.

Syntax

```
SUBSTR({ CharacterExpression },
   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, Derby assumes that you mean 1.)

A *characterExpression* is a CHAR, VARCHAR, or LONG VARCHAR data type or any built-in type that is implicitly converted to a string (except a bit expression).

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*.

SUM

SUM is an aggregate expression that evaluates the sum of the expression over a set of rows (see Aggregates (set functions)). 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. 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 SUM(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. 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 might overflow).

```
-- find all economy seats available:
SELECT SUM (economy_seats) FROM Airlines;

-- use SUM on multiple column references
-- (find the total number of all seats purchased):
SELECT SUM (economy_seats_taken + business_seats_taken +
firstclass_seats_taken)
as seats_taken FROM FLIGHTAVAILABILITY;
```

TIME

The TIME function returns a time from a value. The argument must be a time, timestamp, or a valid string representation of a time or timestamp that is not a CLOB or LONG VARCHAR. The result of the function is a time. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a time: The result is that time.

- If the argument is a timestamp: The result is the time part of the timestamp.
- If the argument is a string: The result is the time represented by the string.

Syntax

```
TIME ( expression )
```

```
values time(current_timestamp)
```

If the current time is 5:03 PM, the value returned is `17:03:00`.

TIMESTAMP

The TIMESTAMP function returns a timestamp from a value or a pair of values.

The rules for the arguments depend on whether the second argument is specified:

- If only one argument is specified: It must be a timestamp, a valid string representation of a timestamp, or a string of length 14 that is not a CLOB or LONG VARCHAR. A string of length 14 must be a string of digits that represents a valid date and time in the form *yyyyxxddhhmmss*, where *yyyy* is the year, *xx* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.
- If both arguments are specified: The first argument must be a date or a valid string representation of a date and the second argument must be a time or a valid string representation of a time.

The other rules depend on whether the second argument is specified:

- If both arguments are specified: The result is a timestamp with the date specified by the first argument and the time specified by the second argument. The microsecond part of the timestamp is zero.
- If only one argument is specified and it is a timestamp: The result is that timestamp.
- If only one argument is specified and it is a string: The result is the timestamp represented by that string. If the argument is a string of length 14, the timestamp has a microsecond part of zero.

Syntax

```
TIMESTAMP ( expression [, expression ] )
```

Assume the column START_DATE (date) has a value equivalent to 1988-12-25, and the column START_TIME (time) has a value equivalent to 17.12.30.

```
TIMESTAMP(START_DATE, START_TIME)
```

Returns the value '1988-12-25-17.12.30.000000'.

UCASE or UPPER

UCASE or UPPER takes a character expression as a parameter and returns a string in which all alpha characters have been converted to uppercase.

Format

```
UCASE or UPPER ( CharacterExpression )
```

If the parameter type is CHAR , the return type is CHAR. Otherwise, the return type is

VARCHAR.
Note: UPPER and LOWER follow the database locale. See *territory=ll_CC* for more information about specifying locale.

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

```
-- returns 'ASD1#W'
VALUES 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
```

```
VALUES USER
```

VARCHAR

The VARCHAR function returns a varying-length character string representation of a character string.

Character to varchar syntax

```
VARCHAR (CharacterStringExpression )
```

CharacterStringExpression
An expression whose value must be of a character-string data type with a maximum length of 32,672 bytes.

Datetime to varchar syntax

```
VARCHAR (DatetimeExpression )
```

DatetimeExpression
An expression whose value must be of a date, time, or timestamp data type.

Using the EMPLOYEE table, select the job description (JOB defined as CHAR(8)) for Dolores Quintana as a VARCHAR equivelant:

```
SELECT VARCHAR(JOB)
FROM EMPLOYEE
WHERE LASTNAME = 'QUINTANA'
```

YEAR

The YEAR function returns the year part of a value. The argument must be a date, timestamp, or a valid character string representation of a date or timestamp. The result of

the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp: The result is the year part of the value, which is an integer between 1 and 9 999.
- If the argument is a date duration or timestamp duration: The result is the year part of the value, which is an integer between -9 999 and 9 999. A nonzero result has the same sign as the argument.

Syntax

```
YEAR ( expression )
```

Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.

```
SELECT * FROM PROJECT
WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
```

Built-in system functions

This section describes the different built-in system functions available with Derby.

SYSCS_UTIL.SYSCS_CHECK_TABLE

The SYSCS_UTIL.SYSCS_CHECK_TABLE function checks the specified table, ensuring that all of its indexes are consistent with the base table. When tables are consistent, the method returns a SMALLINT with value 1. If the tables are inconsistent, the function will throw an exception.

Syntax

```
SMALLINT SYSCS_UTIL.SYSCS_CHECK_TABLE(IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128))
```

An error will occur if either SCHEMANAME or TABLENAME are null.

Example

```
VALUES SYSCS_UTIL.SYSCS_CHECK_TABLE('SALES', 'ORDERS');
```

SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS

The SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS function returns a VARCHAR(32762) value representing the query execution plan and run time statistics for a java.sql.ResultSet. A query execution plan is a tree of execution nodes. There are a number of possible node types. Statistics are accumulated during execution at each node. The types of statistics include the amount of time spent in specific operations, the number of rows passed to the node by its children, and the number of rows returned by the node to its parent. (The exact statistics are specific to each node type.) SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS is most meaningful for DML statements such as SELECT, INSERT, DELETE and UPDATE.

Syntax

```
VARCHAR(32762) SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()
```

Example

```
VALUES SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()
```

SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY

The SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY function fetches the value of a property specified by KEY of the database on the current connection.

Syntax

```
VARCHAR(32762) SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(IN KEY
VARCHAR(128))
```

An error will be returned if KEY is null.

Example

```
VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('key_value_string');
```

Built-in system procedures

Some built-in procedures are not compatible with SQL syntax used by other relational databases. These procedures can only be used with Derby.

SYSCS_UTIL.SYSCS_COMPRESS_TABLE

Use the SYSCS_UTIL.SYSCS_COMPRESS_TABLE system procedure to reclaim unused, allocated space in a table and its indexes. Typically, unused allocated space exists when a large amount of data is deleted from a table, or indexes are updated. By default, Derby does not return unused space to the operating system. For example, once a page has been allocated to a table or index, it is not automatically returned to the operating system until the table or index is destroyed. SYSCS_UTIL.SYSCS_COMPRESS_TABLE allows you to return unused space to the operating system.

Syntax

```
SYSCS_UTIL.SYSCS_COMPRESS_TABLE (IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128), IN SEQUENTIAL SMALLINT)
```

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a null will result in an error.

TABLENAME
An input argument of type VARCHAR(128) that specifies the table name of the table. The string must exactly match the case of the table name, and the argument of "Fred" will be passed to SQL as the delimited identifier 'Fred'. Passing a null will result in an error.

SEQUENTIAL
A non-zero input argument of type SMALLINT will force the operation to run in sequential mode, while an argument of 0 will force the operation not to run in sequential mode. Passing a null will result in an error.

SQL example

To compress a table called CUSTOMER in a schema called US, using the SEQUENTIAL

option:

```
call SYSCS_UTIL.SYSCS_COMPRESS_TABLE('US', 'CUSTOMER', 1)
```

Java example

To compress a table called CUSTOMER in a schema called US, using the SEQUENTIAL option:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_COMPRESS_TABLE(?, ?, ?)");
cs.setString(1, "US");
cs.setString(2, "CUSTOMER");
cs.setShort(3, (short) 1);
cs.execute();
```

If the SEQUENTIAL parameter is not specified, Derby rebuilds all indexes concurrently with the base table. If you do not specify the SEQUENTIAL argument, this procedure can be memory-intensive and use a lot of temporary disk space (an amount equal to approximately two times the used space plus the unused, allocated space). This is because Derby compresses the table by copying active rows to newly allocated space (as opposed to shuffling and truncating the existing space). The extra space used is returned to the operating system on COMMIT.

When SEQUENTIAL is specified, Derby compresses the base table and then compresses each index sequentially. Using SEQUENTIAL uses less memory and disk space, but is more time-intensive. Use the SEQUENTIAL argument to reduce memory and disk space usage.

SYSCS_UTIL.SYSCS_COMPRESS_TABLE cannot release any permanent disk space back to the operating system until a COMMIT is issued. This means that the space occupied by both the base table and its indexes cannot be released back to the operating system until a COMMIT is issued. (Only the disk space that is temporarily claimed by an external sort can be returned to the operating system prior to a COMMIT.) We recommended you issue the SYSCS_UTIL.SYSCS_COMPRESS_TABLE procedure in auto-commit mode.
Note: This procedure acquires an exclusive table lock on the table being compressed. All statement plans dependent on the table or its indexes are invalidated. For information on identifying unused space, see the *Derby Server and Administration Guide* .

SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE

Use the SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE system procedure to reclaim unused, allocated space in a table and its indexes. Typically, unused allocated space exists when a large amount of data is deleted from a table and there has not been any subsequent inserts to use the space created by the deletes. By default, Derby does not return unused space to the operating system. For example, once a page has been allocated to a table or index, it is not automatically returned to the operating system until the table or index is destroyed. SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE allows you to return unused space to the operating system.

This system procedure can be used to force three levels of in-place compression of a SQL table: PURGE_ROWS, DEFRAGMENT_ROWS, and TRUNCATE_END. Unlike SYSCS_UTIL.SYSCS_COMPRESS_TABLE( ), all work is done in place in the existing table/index.

Syntax

```
SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE(
                IN SCHEMANAME VARCHAR(128),
                IN TABLENAME VARCHAR(128),
                IN PURGE_ROWS SMALLINT,
                IN DEFRAGMENT_ROWS SMALLINT,
                IN TRUNCATE_END SMALLINT )
```

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a null will result in an error.

TABLENAME
An input argument of type VARCHAR(128) that specifies the table name of the table. The string must exactly match the case of the table name, and the argument of "Fred" will be passed to SQL as the delimited identifier 'Fred'. Passing a null will result in an error.

PURGE_ROWS
If PURGE_ROWS is set to a non-zero value, then a single pass is made through the table which will purge committed deleted rows from the table. This space is then available for future inserted rows, but remains allocated to the table. As this option scans every page of the table, its performance is linearly related to the size of the table.

DEFRAGMENT_ROWS
If DEFRAGMENT_ROWS is set to a non-zero value, then a single defragment pass is made which will move existing rows from the end of the table towards the front of the table. The goal of defragmentation is to empty a set of pages at the end of the table which can then be returned to the operating system by the TRUNCATE_END option. It is recommended to only run DEFRAGMENT_ROWS if also specifying the TRUNCATE_END option. The DEFRAGMENT_ROWS option scans the whole table and needs to update index entries for every base table row move, so the execution time is linearly related to the size of the table.

TRUNCATE_END
If TRUNCATE_END is set to a non-zero value, then all contiguous pages at the end of the table will be returned to the operating system. Running the PURGE_ROWS and/or DEFRAGMENT_ROWS options may increase the number of pages affected. This option by itself performs no scans of the table.

SQL example
To compress a table called CUSTOMER in a schema called US, using all available compress options:

```
call SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE('US', 'CUSTOMER', 1, 1, 1);
```

To return the empty free space at the end of the same table, the following call will run much quicker than running all options but will likely return much less space:

```
call SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE('US', 'CUSTOMER', 0, 0, 1);
```

Java example
To compress a table called CUSTOMER in a schema called US, using all available compress options:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_COMPRESS_TABLE(?, ?, ?, ?, ?)");
cs.setString(1, "US");
cs.setString(2, "CUSTOMER");
cs.setShort(3, (short) 1);
cs.setShort(4, (short) 1);
cs.setShort(5, (short) 1);
cs.execute();
```

To return the empty free space at the end of the same table, the following call will run

much quicker than running all options but will likely return much less space:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_COMPRESS_TABLE(?, ?, ?, ?, ?)");
cs.setString(1, "US");
cs.setString(2, "CUSTOMER");
cs.setShort(3, (short) 0);
cs.setShort(4, (short) 0);
cs.setShort(5, (short) 1);
cs.execute();
```

It is recommended that the SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE procedure be issued in autocommit mode.
Note: This procedure acquires an exclusive table lock on the table being compressed. All statement plans dependent on the table or its indexes are invalidated. For information on identifying unused space, see the *Derby Server and Administration Guide* .

SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS

The `SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS()` system procedure turns a connection's runtime statistics on or off. By default, the runtime statistics are turned off. When the `runtimestatistics` attribute is turned on, Derby maintains information about the execution plan for each statement executed within the connection (except for COMMIT) until the attribute is turned off. To turn the `runtimestatistics` attribute off, call the procedure with an argument of zero. To turn the `runtimestatistics` on, call the procedure with any non-zero argument.

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 if a `close()` call is encountered, whichever comes first.

Syntax

```
SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(IN SMALLINT ENABLE)
```

Example

```
-- establish a connection
-- turn on RUNTIMESTATISTIC for connection:
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
-- execute complex query here
-- step through the result sets
-- access runtime statistics information:
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(0);
```

SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING

Statistics timing is an attribute associated with a connection that you turn on and off by using the `SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING` system procedure. Statistics timing is turned off by default. Turn statistics timing on only when the `runtimestatistics` attribute is already on. Turning statistics timing on when the `runtimestatistics` attribute is off has no effect.

Turn statistics timing on by calling this procedure with a non-zero argument. Turn statistics timing off by calling the procedure with a zero argument.

When statistics timing is turned on, Derby tracks the timings of various aspects of the execution of a statement. This information is included in the information returned by the SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS system function. When statistics

timing is turned off, the SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS system function shows all timing values as zero.

Syntax

```
SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(IN SMALLINT ENABLE)
```

Example

To turn the `runtimestatistics` attribute and then the statistics timing attribute on:

```
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
CALL SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1);
```

SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY

Use the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure to set or delete the value of a property of the database on the current connection.

If "VALUE" is not null, then the property with key value "KEY" is set to "VALUE". If "VALUE" is null, then the property with key value "KEY" is deleted from the database property set.

Syntax

```
SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(IN KEY VARCHAR(128),
IN VALUE VARCHAR(32672))
```

This procedure does not return any results.

JDBC example

Set the `derby.locks.deadlockTimeout` property to a value of 10:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(?, ?)");
cs.setString(1, "derby.locks.deadlockTimeout");
cs.setString(2, "10");
cs.execute();
cs.close();
```

SQL example

Set the `derby.locks.deadlockTimeout` property to a value of 10:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
('derby.locks.deadlockTimeout', '10');
```

SYSCS_UTIL.SYSCS_FREEZE_DATABASE

The SYSCS_UTIL.SYSCS_FREEZE_DATABASE system procedure temporarily freezes the database for backup.

Syntax

```
SYSCS_UTIL.SYSCS_FREEZE_DATABASE()
```

No result set is returned by this procedure.

Example

```
String backupdirectory = "c:/mybackups/" + JCalendar.getToday();
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()");
cs.execute();
cs.close();
// user supplied code to take full backup of "backupdirectory"
// now unfreeze the database once backup has completed:
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()");
cs.execute();
cs.close();
```

SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE

The SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE system procedure unfreezes a database after backup.

Syntax

```
SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()
```

No result set is returned by this procedure.

Example

```
String backupdirectory = "c:/mybackups/" + JCalendar.getToday();
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()");
cs.execute();
cs.close();
// user supplied code to take full backup of "backupdirectory"
// now unfreeze the database once backup has completed:
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()");
cs.execute();
cs.close();
```

SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE

The SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE system procedure checkpoints the database by flushing all cached data to disk.

Syntax

```
SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE()
```

No result is returned by this procedure.

JDBC example

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE()");
cs.execute();
cs.close();
```

SQL Example

```
CALL SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE();
```

SYSCS_UTIL.SYSCS_BACKUP_DATABASE

The SYSCS_UTIL.SYSCS_BACKUP_DATABASE system procedure backs up the database to a specified backup directory.

Syntax

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE(IN BACKUPDIR VARCHAR())
```

No result is returned from the procedure.

backupDir
An input argument of type VARCHAR(32672) that specifies the full system path to the database directory to be backed up.

JDBC example

The following example backs up the database to the `c:/backupdir` directory:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE(?)");
cs.setString(1, "c:/backupdir");
cs.execute();
cs.close();
```

SQL example

The following example backs up the database to the `c:/backupdir` directory:

```
CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE('c:/backupdir');
```

SYSCS_UTIL.SYSCS_EXPORT_TABLE

The `SYSCS_UTIL.SYSCS_EXPORT_TABLE` system procedure exports all of the data from a table to an operating system file in a delimited data file format.

Syntax

```
SYSCS_UTIL.SYSCS_EXPORT_TABLE (IN SCHEMANAME  VARCHAR(128),
IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672),
IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1),
IN CODESET VARCHAR(128))
```

No result is returned from the procedure.

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema name of the table. Passing a NULL value will use the default schema name.

TABLENAME
An input argument of type VARCHAR(128) that specifies the name of the table/view from which the data is to be exported. Passing a null will result in an error.

FILENAME
An input argument of type VARCHAR(32672) that specifies the name of the file to which data is to be exported. If the complete path to the file is not specified, the export procedure uses the current directory and the default drive as the destination. If the name of a file that already exists is specified, the export procedure overwrites the contents of the file; it does not append the information. Passing a null will result in an error.

COLUMNDELIMITER
An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER
An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET
An input argument of type VARCHAR(128) that specifies the code set of the data in the exported file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the database code set to the specified code set before writing to the file. Passing a NULL value will write the data in the same code set as the JVM in which it is being executed.

If you create a schema or table name as a non-delimited identifier, you must pass the name to the export procedure using all upper-case characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the export procedure using the same case that was used when it was created.

Example

The following example shows how to export information from the STAFF table in a SAMPLE database to the `myfile.del` file.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE (null, 'STAFF', 'myfile.del', null,
null, null);
```

For more information on exporting, see the *Derby Tools and Utilities Guide* .

SYSCS_UTIL.SYSCS_EXPORT_QUERY

The `SYSCS_UTIL.SYSCS_EXPORT_QUERY` system procedure exports the results of a SELECT statement to an operating system file in a delimited data file format.

Syntax

```
SYSCS_UTIL.SYSCS_EXPORT_QUERY(IN SELECTSTATEMENT VARCHAR(32672),
IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1),
IN CHARACTERDELIMITER CHAR(1), IN CODESET VARCHAR(128))
```

No result is returned from the procedure.

SELECTSTATEMENT
An input argument of type VARCHAR(32672) that specifies the select statement (query) that will return the data to be exported. Passing a NULL value will result in an error.

FILENAME
An input argument of type VARCHAR(32672) that specifies the name of the file to which data is to be exported. If the complete path to the file is not specified, the export procedure uses the current directory and the default drive as the destination. If the name of a file that already exists is specified, the export procedure overwrites the contents of the file; it does not append the information. Passing a null will result in an error.

COLUMNDELIMITER
An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER
An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET
An input argument of type VARCHAR(128) that specifies the code set of the data in the exported file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the database code set to the specified code set before writing to the file. Passing a NULL value will write the data in the same code set as the JVM in which it is being executed.

Example

The following example shows how to export the information about employees in Department 20 from the STAFF table in the SAMPLE database to the `myfile.del` file.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_QUERY('select * from staff where dept =20',
'c:/output/awards.del', null, null, null);
```

For more information on exporting, see the *Derby Tools and Utilities Guide* .

SYSCS_UTIL.SYSCS_IMPORT_TABLE

The `SYSCS_UTIL.SYSCS_IMPORT_TABLE` system procedure imports data from an input file into all of the columns of a table. If the table receiving the imported data already contains data, you can either replace or append to the existing data.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_TABLE (IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672),
IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1),
IN CODESET VARCHAR(128), IN REPLACE SMALLINT)
```

No result is returned from the procedure.

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a NULL value will use the default schema name.

TABLENAME
An input argument of type VARCHAR (128) that specifies the table name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. Passing a null will result in an error.

FILENAME
An input argument of type VARCHAR(32672) that specifies the file that contains the data to be imported. If you do not specify a path, the current working directory is used. Passing a NULL value will result in an error.

COLUMNDELIMITER
An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER
An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET
An input argument of type VARCHAR(128) that specifies the code set of the data in the input file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the specified code set to the database code set (utf-8). Passing a NULL value will interpret the data file in the same code set as the JVM in which it is being executed.

REPLACE
A input argument of type SMALLINT. A non-zero value will run in REPLACE mode, while a value of zero will run in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the data object, and inserts the imported data. The table definition and the index definitions are not changed. INSERT mode adds the imported data to the table without changing the existing table data. Passing a NULL will result in an error.

If you create a schema, table, or column name as a non-delimited identifier, you must

pass the name to the import procedure using all upper-case characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the import procedure using the same case that was used when it was created.

Example
The following example imports data into the `staff` table from a delimited data file called `myfile.del` with the percentage character (%) as the string delimiter, and a semicolon (;) as the column delimiter:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE
(null, 'STAFF', 'c:/output/myfile.del', ';', '%', null,0);
```

For more information on importing, see the *Derby Tools and Utilities Guide* .

SYSCS_UTIL.SYSCS_IMPORT_DATA

The `SYSCS_UTIL.SYSCS_IMPORT_DATA` system procedure imports data to a subset of columns in a table. You choose the subset of columns by specifying insert columns. This procedure is also used to import a subset of column data from a file by specifying column indexes.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_DATA (IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128), IN INSERTCOLUMNS VARCHAR(32672),
IN COLUMNINDEXES VARCHAR(32672), IN FILENAME VARCHAR(32672),
IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1),
IN CODESET VARCHAR(128), IN REPLACE SMALLINT)
```

No result is returned from the procedure.

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a NULL value will use the default schema name.

TABLENAME
An input argument of type VARCHAR (128) that specifies the table name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. Passing a null will result in an error.

INSERTCOLUMNS
An input argument of type VARCHAR (32762) that specifies the column names (separated by commas) of the table into which the data is to be imported. Passing a NULL value will import the data into all of the columns of the table.

COLUMNINDEXES
An input argument of type VARCHAR (32762) that specifies the indexes (numbered from 1 and separated by commas) of the input data fields to be imported. Passing a NULL value will use all of the input data fields in the file.

FILENAME
An input argument of type VARCHAR(32672) that specifies the file that contains the data to be imported. If you do not specify a path, the current working directory is used. Passing a NULL value will result in an error.

COLUMNDELIMITER
An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER
An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET

An input argument of type VARCHAR(128) that specifies the code set of the data in the input file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the specified code set to the database code set (utf-8). Passing a NULL value will interpret the data file in the same code set as the JVM in which it is being executed.

REPLACE

A input argument of type SMALLINT. A non-zero value will run in REPLACE mode, while a value of zero will run in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the data object, and inserts the imported data. The table definition and the index definitions are not changed. You can only use the REPLACE mode if the table exists. INSERT mode adds the imported data to the table without changing the existing table data. Passing a NULL will result in an error.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the import procedure using all upper-case characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the import procedure using the same case that was used when it was created.

Example

The following example imports some of the data fields from a delimited data file called `data.del` into the `staff` table:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA
(NULL, 'STAFF', null, '1,3,4', 'data.del', null, null, null,0)
```

For more information on importing, see the *Derby Tools and Utilities Guide* .

Data types

This section describes the data types used in Derby.

Built-In type overview

The SQL 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.

Numeric types

Numeric types used in Derby.

Numeric type overview

Numeric types include the following types, which provide storage of varying sizes:

- Integer numerics
  - SMALLINT (2 bytes)
  - INTEGER (4 bytes)
  - BIGINT (8 bytes)
- Approximate or floating-point numerics
  - 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, Derby promotes the type of the result to at least INTEGER. In expressions that mix integer with non-integer types, Derby promotes the result of the expression to the highest type in the expression. Type Promotion in Expressions shows the promotion of data types in expressions.

Table1. Type Promotion in Expressions

Largest Type That Appears in Expression	Resulting Type of Expression
DOUBLE PRECISION	DOUBLE PRECISION
REAL	DOUBLE PRECISION
DECIMAL	DECIMAL
BIGINT	BIGINT
INTEGER	INTEGER
SMALLINT	INTEGER

For example:

```
-- returns a double precision
VALUES 1 + 1.0e0
-- returns a decimal
VALUES 1 + 1.0
-- returns an integer
VALUES CAST (1 AS INT) + CAST (1 AS INT)
```

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:

```
create table mytable (r REAL, d DOUBLE PRECISION);
0 rows inserted/updated/deleted
INSERT INTO mytable (r, d) values (3.4028236E38, 3.4028235E38);
ERROR X0X41: The number '3.4028236E38' is outside the range for
the data type REAL.
```

You can store a floating point type in an INTEGER column; the fractional part of the number is truncated. For example:

```
INSERT INTO mytable(integer_column) values (1.09e0);
1 row inserted/updated/deleted
SELECT integer_column
FROM mytable;
I
--------------
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:

```
ij> 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:

```
INSERT INTO mytable (int_column) values 2147483648;
ERROR 22003: The resulting value is outside the range for the
data type INTEGER.
```

Note: When truncating trailing digits from a NUMERIC value, Derby rounds down.

Scale for decimal arithmetic

SQL statements can involve arithmetic expressions that use decimal data types of different *precision*s (the total number of digits, both to the left and to the right of the decimal point) and *scale*s (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*

  31 - lp + ls - rs

- *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 27:

```
11.0/1111.33
// 31 - 3 + 1 - 2 = 27
```

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)

Data type assignments and comparison, sorting, and ordering

Table1. Assignments allowed by Derby

This table displays valid assignments between data types in Derby. A "Y" indicates that the assignment is valid.

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP
SMALL INT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
FLOAT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
CHAR	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y
VARCHAR	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y
LONG VARCHAR	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-
CHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	-	-
VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	-	-
LONG VARCHAR	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	-	-

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP
FOR BIT DATA																		
CLOB	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-
DATE	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-
TIME	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-
TIME STAMP	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	Y

Table1. Comparisons allowed by Derby
This table displays valid comparisons between data types in Derby. A "Y" indicates that the comparison is allowed.

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP
SMALL INT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
FLOAT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
CHAR	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	Y	Y
VARCHAR	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	Y	Y
LONG VARCHAR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-
VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-
LONG VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DATE	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-
TIME	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-
TIME STAMP	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	Y

BIGINT

BIGINT provides 8 bytes of storage for integer values.

Syntax

```
BIGINT
```

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 .

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. BIGINTs can be stored in DECIMALs if the DECIMAL precision is large enough for the value.

```
9223372036854775807
```

BLOB

A BLOB (binary large object) is a varying-length binary string that can be up to 2,147,483,647 characters long. Like other binary types, BLOB strings are not associated with a code page. In addition, BLOB strings do not hold character data.

The length is given in bytes for BLOB unless one of the suffixes K, M, or G is given, relating to the multiples of 1024, 1024*1024, 1024*1024*1024 respectively.

Note: Length is specified in bytes for BLOB.

Syntax

```
{ BLOB | BINARY LARGE OBJECT } ( length [{K |M |G }]))
```

Corresponding compile-time Java type

java.sql.Blob

JDBC metadata type (java.sql.Types)

BLOB

Use the *getBlob* method on the *java.sql.ResultSet* to retrieve a BLOB handle to the

underlying data.

Related information

see java.sql.Blob and java.sql.Clob

```
create table pictures(name varchar(32) not null primary key, pic
blob(16M));

--find all logotype pictures
select length(pic), name from pictures where name like '%logo%';

--find all image doubles (blob comparsions)
select a.name as double_one, b.name as double_two
from pictures as a, pictures as b
where a.name < b.name
and a.pic = b.pic
order by 1,2;
```

CHAR

CHAR provides for fixed-length storage of strings.

Syntax

```
CHAR[ACTER] [(length)]
```

length is an unsigned integer constant. The default length for a CHAR is 1.

Corresponding compile-time Java type

java.lang.String

JDBC metadata type (java.sql.Types)

CHAR

Derby inserts spaces to pad a string value shorter than the expected length. Derby 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 constant is CHAR.

Implementation-defined aspects

The only limit on the length of CHAR data types is the value *java.lang.Integer.MAX_VALUE*.

```
-- within a string constant use two single quotation marks
-- to represent a single quotation mark or apostrophe
VALUES 'hello this is Joe''s string'
```

CHAR FOR BIT DATA

A CHAR FOR BIT DATA type allows you to store byte strings of a specified length. It is useful for unstructured data where character strings are not appropriate.

Syntax

```
{ CHAR | CHARACTER }[(length)] FOR BIT DATA
```

length is an unsigned integer literal designating the length in bytes.

The default *length* for a CHAR FOR BIT DATA type is 1., and the maximum size of *length* is 254 bytes.

JDBC metadata type (java.sql.Types)

BINARY

CHAR FOR BIT DATA stores fixed-length byte strings. If a CHAR FOR BIT DATA value is smaller than the target CHAR FOR BIT DATA, it is padded with a 0x20 byte value.

Comparisons of CHAR FOR BIT DATA and VARCHAR FOR BIT DATA 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 VARCHAR FOR BIT DATA and a CHAR FOR BIT DATA value (e.g., a concatenation) yields a VARCHAR FOR BIT DATA value.

```
CREATE TABLE t (b CHAR(2) FOR BIT DATA);
INSERT INTO t VALUES (X'DE');
SELECT *
FROM t;
-- yields the following output
B
-----
de20
```

CLOB

A CLOB (character large object) value can be up to 2,147,483,647 characters long. A CLOB is used to store unicode character-based data, such as large documents in any character set.

The length is given in number characters for both CLOB, unless one of the suffixes K, M, or G is given, relating to the multiples of 1024, 1024*1024, 1024*1024*1024 respectively.

Length is specified in characters (unicode) for CLOB.

Syntax

```
{CLOB |CHARACTER LARGE OBJECT}(length [{{K |M |G}]))
```

Corresponding Compile-Time Java Type

java.sql.Clob

JDBC Metadata Type (java.sql.Types)

CLOB

Corresponding Compile-Time Java Type

java.sql.Clob

JDBC Metadata Type (java.sql.Types)

CLOB

Use the *getClob* method on the *java.sql.ResultSet* to retrieve a CLOB handle to the underlying data.

Related Information

See java.sql.Blob and java.sql.Clob .

```
import java.sql.*;

public class clob
{
        public static void main(String[] args) {
                try {
                        String url =
"jdbc:derby:clobberyclob;create=true";

Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
                        Connection conn =
DriverManager.getConnection(url);

            Statement s = conn.createStatement();
            s.executeUpdate("CREATE TABLE documents (id INT, text CLOB(64
K))");
            conn.commit();

            // --- add a file
            java.io.File file = new java.io.File("asciifile.txt");
            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 documents VALUES (?, ?)");
            ps.setInt(1, 1477);

            // - set the value of the input parameter to the input stream
            ps.setAsciiStream(2, fin, fileLength);
            ps.execute();
            conn.commit();

            // --- reading the columns
            ResultSet rs = s.executeQuery("SELECT text FROM documents
             WHERE id = 1477");
            while (rs.next()) {
                java.sql.Clob aclob = rs.getClob(1);
                java.io.InputStream ip = rs.getAsciiStream(1);
                int c = ip.read();
                while (c > 0) {
                    System.out.print((char)c);
                    c = ip.read();
                }
                System.out.print("\n");
                // ...
            }
        } catch (Exception e) {
            System.out.println("Error! "+e);
        }
    }
}
```

DATE

DATE provides for storage of a year-month-day in the range supported by *java.sql.Date*.

Syntax

```
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.

Any value that is recognized by the *java.sql.Date* method is permitted in a column of the corresponding SQL date/time data type. Derby supports the following formats for DATE:

```
yyyy-mm-dd
mm/dd/yyyy
dd.mm.yyyy
```

The first of the three formats above is the *java.sql.Date* format.

The year must always be expressed with four digits, while months and days may have either one or two digits.

Derby also accepts strings in the locale specific datetime format, using the locale of the database server. If there is an ambiguity, the built-in formats above take precedence.

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

```
{ DECIMAL | DEC } [(precision [, scale ])]
```

The *precision* must be between 1 and 31. 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 5.

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, Derby rounds down.

For example:

```
-- this cast loses only fractional precision
values cast (1.798765 AS decimal(5,2));
1
--------
1.79
```

```
-- this cast does not fit
values cast (1798765 AS decimal(5,2));
1
--------
ERROR 22003: The resulting value is outside the range
for the data type DECIMAL/NUMERIC(5,2).
```

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions .

See also Storing values of one numeric data type in columns of another numeric data type .

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 .

Corresponding compile-time Java type

java.math.BigDecimal

JDBC metadata type (java.sql.Types)

DECIMAL

```
VALUES 123.456
VALUES 0.001
```

Integer constants too big for BIGINT are made DECIMAL constants.

DOUBLE

The DOUBLE data type is a synonym for the DOUBLE PRECISION data type.

Syntax

```
DOUBLE
```

DOUBLE PRECISION

The DOUBLE PRECISION data type provides 8-byte storage for numbers using IEEE floating-point notation.

Syntax

```
DOUBLE PRECISION
```

or, alternately

```
DOUBLE
```

DOUBLE can be used synonymously with DOUBLE PRECISION.

Limitations

DOUBLE value ranges:

- Smallest DOUBLE value: -1.79769E+308
- Largest DOUBLE value: 1.79769E+308
- Smallest positive DOUBLE value: 2.225E-307
- Largest negative DOUBLE value: -2.225E-307

These limits are different from the `java.lang.Double`Java type limits.

An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations **do not** round their resulting values to zero. If the values are too small, you will receive an exception.

Numeric floating point constants are limited to a length of 30 characters.

```
-- this example will fail because the constant is too long:
values 0123456789012345678901234567890e0;
```

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 .

See also Storing values of one numeric data type in columns of another numeric data type .

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

```
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.

JDBC metadata type (java.sql.Types)

REAL or DOUBLE

Limitations

If you are using a precision of 24 or greater, the limits of FLOAT are similar to the limits of DOUBLE.

If you are using a precision of 23 or less, the limits of FLOAT are similar to the limits of

REAL.

INTEGER

INTEGER provides 4 bytes of storage for integer values.

Syntax

```
{ 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 .

See also Storing values of one numeric data type in columns of another numeric data type .

```
3453
425
```

LONG VARCHAR

The LONG VARCHAR type allows storage of character strings with a maximum length of 32,700 characters. It is identical to VARCHAR, except that you do not have to specify a maximum length when creating columns of this type.

Syntax

```
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.

LONG VARCHAR FOR BIT DATA

The LONG VARCHAR FOR BIT DATA type allows storage of bit strings up to 32,700 bytes. It is identical to VARCHAR FOR BIT DATA , except that you do not have to specify a maximum length when creating columns of this type.

Syntax

```
LONG VARCHAR FOR BIT DATA
```

NUMERIC

NUMERIC is a synonym for DECIMAL and behaves the same way. See DECIMAL .

Syntax

```
NUMERIC [(precision [, scale ])]
```

Corresponding compile-time Java type

java.math.BigDecimal

JDBC metadata Ttype (java.sql.Types)

NUMERIC

```
123.456
.001
```

REAL

The REAL data type provides 4 bytes of storage for numbers using IEEE floating-point notation.

Syntax

```
REAL
```

Corresponding compile-time Java type

java.lang.Float

JDBC metadata type (java.sql.Types)

REAL

Limitations

REAL value ranges:

- Smallest REAL value: -3.402E+38
- Largest REAL value: 3.402E+38
- Smallest positive REAL value: 1.175E-37
- Largest negative REAL value: -1.175E-37

These limits are different from the `java.lang.Float` Java type limits.

An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations **do not** round their resulting values to zero. If

the values are too small, you will receive an exception. The arithmetic operations take place with double arithmetic in order to detect under flows.

Numeric floating point constants are limited to a length of 30 characters.

```
-- this example will fail because the constant is too long:
values 0123456789012345678901234567890e0;
```

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions .

See also Storing values of one numeric data type in columns of another numeric data type .

Constants always map to DOUBLE PRECISION; use a CAST to convert a constant to a REAL.

SMALLINT

SMALLINT provides 2 bytes of storage.

Syntax

```
SMALLINT
```

Corresponding compile-time Java type

java.lang.Short

JDBC metadata type (java.sql.Types)

SMALLINT

Minimum value

-32768 (*java.lang.Short.MIN_VALUE*)

Maximum value

32767 (*java.lang.Short.MAX_VALUE*)

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions .

See also Storing values of one numeric data type in columns of another numeric data type .

Constants in the appropriate format always map to INTEGER or BIGINT, depending on their length.

TIME

TIME provides for storage of a time-of-day value.

Syntax

```
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.

Any value that is recognized by the *java.sql.Time* method is permitted in a column of the corresponding SQL date/time data type. Derby supports the following formats for TIME:

```
hh:mm[:ss]
hh.mm[.ss]
hh[:mm] {AM | PM}
```

The first of the three formats above is the *java.sql.Time* format.

Hours may have one or two digits. Minutes and seconds, if present, must have two digits.

Derby also accepts strings in the locale specific datetime format, using the locale of the database server. If there is an ambiguity, the built-in formats above take precedence.

Examples

```
VALUES TIME('15:09:02')
VALUES '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

```
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.

Derby supports the following formats for TIMESTAMP:

```
yyyy-mm-dd hh[:mm[:ss[.nnnnnn]]]
yyyy-mm-dd-hh[.mm[.ss[.nnnnnn]]]
```

The first of the two formats above is the *java.sql.Timestamp* format.

The year must always have four digits. Months, days, and hours may have one or two digits. Minutes and seconds, if present, must have two digits. Nanoseconds, if present may have between one and six digits.

Derby also accepts strings in the locale specific datetime format, using the locale of the database server. If there is an ambiguity, the built-in formats above take precedence.

Examples

```
VALUES '1960-01-01 23:03:20'
VALUES TIMESTAMP('1962-09-23 03:23:34.234')
VALUES TIMESTAMP('1960-01-01 23:03:20')
```

VARCHAR

VARCHAR provides for variable-length storage of strings.

Syntax

```
{ VARCHAR | CHAR VARYING | CHARACTER VARYING }(length)
```

length is an unsigned integer constant, and it must not be greater than the constraint of the integer used to specify the length, the value *java.lang.Integer.MAX_VALUE*.

The maximum length for a VARCHAR string is 32,672 characters.

Corresponding compile-time Java type

java.lang.String

JDBC metadata type (java.sql.Types)

VARCHAR

Derby does not pad a VARCHAR value whose length is less than specified. Derby 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 constant is CHAR, not VARCHAR.

VARCHAR FOR BIT DATA

The VARCHAR FOR BIT DATA type allows you to store binary strings less than or equal to a specified length. It is useful for unstructured data where character strings are not appropriate (e.g., images).

Syntax

```
{ VARCHAR | CHAR VARYING | CHARACTER VARYING } (length) FOR BIT DATA
```

length is an unsigned integer literal designating the length in bytes.

Unlike the case for the CHAR FOR BIT DATA type, there is no default *length* for a VARCHAR FOR BIT DATA type. The maximum size of the *length* value is 32,672 bytes.

JDBC metadata type (java.sql.Types)

VARBINARY

VARCHAR FOR BIT DATA stores variable-length byte strings. Unlike CHAR FOR BIT DATA values, VARCHAR FOR BIT DATA values are not padded out to the target length.

An operation on a VARCHAR FOR BIT DATA and a CHAR FOR BIT DATA value (e.g., a concatenation) yields a VARCHAR FOR BIT DATA value.

The type of a byte literal is always a VARCHAR FOR BIT DATA, not a CHAR FOR BIT DATA.

SQL expressions

Syntax for many statements and expressions includes the term *Expression*, or a term for a specific kind of expression such as *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 of Expressions , lists all the possible SQL 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.

Table1. Table of Expressions

'	Expression Type	Explanation
General expressions		
'	Column reference Allowed in *SelectExpression* s, UPDATE statements, and the WHERE clauses of data manipulation statements.	A *column-Name* that references the value of the column made visible to the expression containing the Column reference. You must qualify the *column-Name* by the table name or correlation name if it is ambiguous. The qualifier of a *column-Name* 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 *column-Name* qualifier once it has been aliased by a correlation name.
'	Constant	Most built-in data types typically have constants associated with them (as shown in Data types).
'	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 constant representing the unknown value.
'	Dynamic parameter	A dynamic parameter is a parameter to an SQL statement for which the value is not specified when

'	Expression Type	Explanation
	Allowed anywhere in an expression where the data type can be easily deduced. See Dynamic parameters .	the statement is created. Instead, the statement has a question mark (?) as a placeholder for each dynamic parameter. See Dynamic parameters . 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	Lets you specify the type of NULL or of a dynamic parameter or convert a value to another type. See CAST .
'	scalar subquery	Subquery that returns a single row with a single column. See *ScalarSubquery* .
'	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* .
'	Conditional expression	A conditional expression chooses an expression to evaluate based on a boolean test.
Boolean expressions		
Numeric expressions		
'	+, -, *, /, unary + and - expressions	+, -, *, /, unary + and - 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 . 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.
'	AVG	Returns the average of a set of numeric values. AVG
'	SUM	Returns the sum of a set of numeric values. SUM
'	LENGTH	Returns the number of characters in a character or bit string. See LENGTH .
'	LOWER	See LCASE or LOWER .
'	COUNT	Returns the count of a set of values. See COUNT , COUNT(*) .
Character expressions		
'	A CHAR or VARCHAR value that uses	The wildcards % and _ make a character string a pattern against which the LIKE operator can look for

'	Expression Type	Explanation
	wildcards. Used in a LIKE pattern.	a match.
'	Concatenation expression	In a concatenation expression, the concatenation operator, "||", concatenates its right operand to the end of its left operand. Operates on character and bit strings. See Concatenation .
'	Built-in string functions	The built-in string functions act on a String and return a string. See LTRIM , LCASE or LOWER , RTRIM , SUBSTR , and UCASE or UPPER
'	USER functions	User functions return information about the current user as a String. See CURRENT_USER , SESSION_USER , and USER
Date/time expressions		
'	CURRENT_DATE	Returns the current date. See CURRENT_DATE .
'	CURRENT_TIME	Returns the current time. See CURRENT_TIME .
'	CURRENT_TIMESTAMP	Returns the current timestamp. See CURRENT_TIMESTAMP .

Expression precedence

Precedence of operations from highest to lowest is:

- (), ?, Constant (including sign), NULL, *ColumnReference*, *ScalarSubquery*, CAST
- LENGTH, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, and other built-ins
- unary + and -
- *, /, || (concatenation)
- binary + and -
- comparisons, quantified comparisons, EXISTS, IN, IS NULL, LIKE, BETWEEN, IS
- NOT
- AND
- OR

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 OR age > 65) AND employed = TRUE
```

Boolean expression

Boolean expressions are allowed in WHERE clauses and in check constraints. Boolean expressions in check constraints have limitations not noted here; see CONSTRAINT clause for more information. Boolean expressions in a WHERE clause have a highly

liberal syntax; see WHERE clause , for example.

A boolean expression can include a boolean operator or operators. These are listed in SQL Boolean Operators .

Table1. SQL Boolean Operators

Operator	Explanation and Example	Syntax
AND, OR, NOT	Evaluate any operand(s) that are boolean expressions `(orig_airport = 'SFO') OR (dest_airport = 'GRU') -- returns true`	`{ Expression AND Expression | Expression OR Expression | NOT Expression }`
Comparisons	<, =, >, <=, >=, <> are applicable to all of the built-in types. `DATE('1998-02-26') < DATE('1998-03-01') -- returns true`	`Expression { < | = | > | <= | >= | <> } Expression`
IS NULL, IS NOT NULL	Test whether the result of an expression is null or not. `WHERE MiddleName IS NULL`	`Expression IS [ NOT ] NULL`
LIKE	Attempts to match a character expression to a character pattern, which is a character string that includes one or more wildcards. % matches any number (zero or more) of characters in the corresponding position in first character expression. _ matches one character in the corresponding position in the character expression. Any other character matches only that character in the corresponding position in the character expression. `city LIKE 'Sant_'` To treat % or _ as constant characters, escape	`CharacterExpression [ NOT ] LIKE CharacterExpression WithWildCard [ ESCAPE ' escapeCharacter ']`

Operator	Explanation and Example	Syntax
	the character with an optional escape character, which you specify with the ESCAPE clause. `SELECT a FROM tabA WHERE a LIKE '%=_' ESCAPE '='`	
BETWEEN	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 <= and >= can be applied. `WHERE booking_date BETWEEN DATE('1998-02-26') AND DATE('1998-03-01')`	`Expression [ NOT ] BETWEEN Expression AND Expression`
IN	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. `WHERE booking_date NOT IN (SELECT booking_date FROM HotelBookings WHERE rooms_available = 0)`	`{ Expression [ NOT ] IN TableSubquery | Expression [ NOT ] IN ( Expression [, Expression ]* ) }`
EXISTS	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. `WHERE EXISTS (SELECT * FROM Flights WHERE dest_airport = 'SFO' AND orig_airport = 'GRU')`	`[NOT] EXISTS TableSubquery`
Quantified comparison	A quantified comparison is a comparison operator (<, =, >, <=, >=, <>) 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	`Expression ComparisonOperator { ALL | ANY | SOME } TableSubquery`

Operator	Explanation and Example	Syntax
	subquery. ANY and SOME are equivalent. `WHERE normal_rate < ALL (SELECT budget/550 FROM Groups)`	

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, they are always IN parameters.

New: Derby supports the interface *ParameterMetaData*, new in JDBC 3.0. This interface describes the number, type, and properties of prepared statement parameters. See the *Derby Developer's Guide* for more information.

You must specify values for them before executing the statement. The values specified must match the types expected.

Dynamic parameters example

```
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 the previous rule. 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 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 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 CHAR FOR BIT DATA or VARCHAR FOR BIT DATA type, the type of the parameter is VARCHAR FOR BIT DATA with the maximum allowed length for the type.

```
SELECT BITcolumn || ?
FROM UserTable
-- Type assumed to be CHAR FOR BIT DATA 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).

```
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. 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, INTERSECT, or EXCEPT (see no. 16 , 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
```

11. 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.
```

12. 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
```

13. A dynamic parameter is not allowed as the operand of the unary operators - or +.
14. LENGTH allow a dynamic parameter. The type is assumed to be a maximum length VARCHAR type.

```
SELECT LENGTH(?)
```

15. Qualified comparisons.

```
? = SOME (SELECT 1 FROM t)
-- is valid. Dynamic parameter assumed to be INTEGER type
1 = SOME (SELECT ? FROM t)
-- is valid. Dynamic parameter assumed to be INTEGER type.
```

16. A dynamic parameter is allowed to represent a column if it appears in a UNION, INTERSECT, or EXCEPT expression; Derby can infer the data type from the corresponding column in the expression.

```
SELECT ?
FROM t
UNION SELECT 1
FROM t
-- dynamic parameter assumed to be INT
VALUES 1 UNION VALUES ?
-- dynamic parameter assumed to be INT
```

17. 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 INTEGER (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 -.

SQL reserved words

This section lists all the Derby reserved words, including those in the SQL-92 standard. Derby will return an error if you use any of these keywords as an identifier name unless you surround the identifier name with quotes ("). See Rules for SQL92 identifiers .

ADD
ALL
ALLOCATE
ALTER
AND
ANY
ARE
AS
ASC
ASSERTION
AT
AUTHORIZATION
AVG
BEGIN
BETWEEN
BIT
BOOLEAN
BOTH
BY
CALL
CASCADE
CASCADED
CASE
CAST
CHAR
CHARACTER
CHECK
CLOSE
COLLATE
COLLATION
COLUMN
COMMIT
CONNECT
CONNECTION
CONSTRAINT
CONSTRAINTS
CONTINUE
CONVERT
CORRESPONDING
COUNT
CREATE
CURRENT
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
CURSOR
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFERRABLE

DEFERRED
DELETE
DESC
DESCRIBE
DIAGNOSTICS
DISCONNECT
DISTINCT
DOUBLE
DROP
ELSE
END
ENDEXEC
ESCAPE
EXCEPT
EXCEPTION
EXEC
EXECUTE
EXISTS
EXPLAIN
EXTERNAL
FALSE
FETCH
FIRST
FLOAT
FOR
FOREIGN
FOUND
FROM
FULL
FUNCTION
GET
GET_CURRENT_CONNECTION
GLOBAL
GO
GOTO
GRANT
GROUP
HAVING
HOUR
IDENTITY
IMMEDIATE
IN
INDICATOR
INITIALLY
INNER
INOUT
INPUT
INSENSITIVE
INSERT
INT
INTEGER
INTERSECT
INTO
IS
ISOLATION
JOIN
KEY
LAST
LEFT

LIKE
LONGINT
LOWER
LTRIM
MATCH
MAX
MIN
MINUTE
NATIONAL
NATURAL
NCHAR
NVARCHAR
NEXT
NO
NOT
NULL
NULLIF
NUMERIC
OF
ON
ONLY
OPEN
OPTION
OR
ORDER
OUT
OUTER
OUTPUT
OVERLAPS
PAD
PARTIAL
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVILEGES
PROCEDURE
PUBLIC
READ
REAL
REFERENCES
RELATIVE
RESTRICT
REVOKE
RIGHT
ROLLBACK
ROWS
RTRIM
SCHEMA
SCROLL
SECOND
SELECT
SESSION_USER
SET
SMALLINT
SOME
SPACE
SQL
SQLCODE

SQLERROR
SQLSTATE
SUBSTR
SUBSTRING
SUM
SYSTEM_USER
TABLE
TEMPORARY
TIMEZONE_HOUR
TIMEZONE_MINUTE
TO
TRAILING
TRANSACTION
TRANSLATE
TRANSLATION
TRUE
UNION
UNIQUE
UNKNOWN
UPDATE
UPPER
USER
USING
VALUES
VARCHAR
VARYING
VIEW
WHENEVER
WHERE
WITH
WORK
WRITE
XML
XMLEXISTS
XMLPARSE
XMLSERIALIZE
YEAR

Derby support for SQL-92 features

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

Table1. Support for SQL-92 Features: Basic types

Feature	Source	Derby
SMALLINT	SQL92E	yes
INTEGER	SQL92E	yes
DECIMAL(p,s)	SQL92E	yes
NUMERIC(p,s)	SQL92E	yes
REAL	SQL92E	yes
FLOAT(p)	SQL92E	yes
DOUBLE PRECISION	SQL92E	yes
CHAR(n)	SQL92E	yes

Table1. Support for SQL-92 Features: Basic math operations

Feature	Source	Derby
+, *, -, /, unary +, unary -	SQL92E	yes

Table1. Support for SQL-92 Features: Basic comparisons

Feature	Source	Derby
<, >, <= ,>=, <>, =	SQL92E	yes

Table1. Support for SQL-92 Features: Basic predicates

Feature	Source	Derby
BETWEEN, LIKE, NULL	SQL92E	yes

Table1. Support for SQL-92 Features: Quantified predicates

Feature	Source	Derby
IN, ALL/SOME, EXISTS	SQL92E	yes

Table1. Support for SQL-92 Features: schema definition

Feature	Source	Derby
tables	SQL92E	yes
views	SQL92E	yes
privileges	SQL92E	yes

Table1. Support for SQL-92 Features: column attributes

Feature	Source	Derby
default values	SQL92E	yes
nullability	SQL92E	yes

Table1. Support for SQL-92 Features: constraints (non-deferrable)

Feature	Source	Derby
NOT NULL	SQL92E	yes (not stored in SYSCONSTRAINTS)
UNIQUE/PRIMARY KEY	SQL92E	yes
FOREIGN KEY	SQL92E	yes
CHECK	SQL92E	yes
View WITH CHECK OPTION	SQL92E	no, since views are not updatable

Table1. Support for SQL-92 Features: Cursors

Feature	Source	Derby
DECLARE, OPEN, FETCH, CLOSE	SQL92E	done through JDBC
UPDATE, DELETE CURRENT	SQL92E	yes

Table1. Support for SQL-92 Features: Dynamic SQL 1

Feature	Source	Derby
ALLOCATE / DEALLOCATE / GET / SET DESCRIPTOR	SQL92T	done through JDBC
PREPARE / EXECUTE / EXECUTE IMMEDIATE	SQL92T	done through JDBC
DECLARE, OPEN, FETCH, CLOSE, UPDATE, DELETE dynamic cursor	SQL92T	done through JDBC
DESCRIBE output	SQL92T	done through JDBC

Table1. Support for SQL-92 Features: Basic information schema

Feature	Source	Derby
TABLES	SQL92T	*SYS.SYSTABLES, SYS.SYSVIEWS, SYS.SYSCOLUMNS*
VIEWS	SQL92T	*SYS.SYSTABLES,*

Feature	Source	Derby
		SYS.SYSVIEWS, SYS.SYSCOLUMNS
COLUMNS	SQL92T	*SYS.SYSTABLES, SYS.SYSVIEWS, SYS.SYSCOLUMNS*

Table1. Support for SQL-92 Features: Basic schema manipulation

Feature	Source	Derby
CREATE / DROP TABLE	SQL92T	yes
CREATE / DROP VIEW	SQL92T	yes
GRANT / REVOKE	SQL92T	no
ALTER TABLE ADD COLUMN	SQL92T	yes
ALTER TABLE DROP COLUMN	SQL92T	no

Table1. Support for SQL-92 Features: Joined table

Feature	Source	Derby
INNER JOIN	SQL92T	yes
natural join	SQL92T	no
LEFT, RIGHT OUTER JOIN	SQL92T	yes
join condition	SQL92T	yes
named columns join	SQL92T	yes

Table1. Support for SQL-92 Features: Joined table

Feature	Source	Derby
simple DATE, TIME, TIMESTAMP, INTERVAL	SQL92T	yes, not INTERVAL
datetime constants	SQL92T	yes
datetime math	SQL92T	can do with Java methods
datetime comparisons	SQL92T	yes
predicates: OVERLAPS	SQL92T	can do with Java methods

Table1. Support for SQL-92 Features: VARCHAR

Feature	Source	Derby
LENGTH	SQL92T	yes
concatenation (||)	SQL92T	yes

Table1. Support for SQL-92 Features: Transaction isolation

Feature	Source	Derby
READ WRITE / READ ONLY	SQL92T	through JDBC, database properties, and storage media.
RU, RC, RR, SER	SQL92T	yes

Table1. Support for SQL-92 Features: Multiple schemas per user

Feature	Source	Derby
SCHEMATA view	SQL92T	*SYS.SYSSCHEMAS*

Table1. Support for SQL-92 Features: Privilege tables

Feature	Source	Derby
TABLE_PRIVILEGES	SQL92T	no
COLUMNS_PRIVILEGES	SQL92T	no
USAGE_PRIVILEGES	SQL92T	no

Table1. Support for SQL-92 Features: Table operations

Feature	Source	Derby
UNION relaxations	SQL92I	yes
EXCEPT	SQL92I	yes
INTERSECT	SQL92I	yes
CORRESPONDING	SQL92I	no

Table1. Support for SQL-92 Features: Schema definition statement

Feature	Source	Derby
CREATE SCHEMA	SQL92I	yes, partially

Table1. Support for SQL-92 Features: User authorization

Feature	Source	Derby
SET SESSION AUTHORIZATION	SQL92I	use set schema
CURRENT_USER	SQL92I	yes
SESSION_USER	SQL92I	yes
SYSTEM_USER	SQL92I	no

Table1. Support for SQL-92 Features: Constraint tables

Feature	Source	Derby
TABLE CONSTRAINTS	SQL92I	*SYS.SYSCONSTRAINTS*
REFERENTIAL CONSTRAINTS	SQL92I	*SYS.SYSFOREIGNKEYS*
CHECK CONSTRAINTS	SQL92I	*SYS.SYSCHECKS*

Table1. Support for SQL-92 Features: Documentation schema

Feature	Source	Derby
SQL_FEATURES	SQL92I/FIPS 127-2	use JDBC *DatabaseMetaData*
SQL_SIZING	SQL92I/FIPS 127-2	use JDBC *DatabaseMetaData*

Table1. Support for SQL-92 Features: Full DATETIME

Feature	Source	Derby
precision for TIME and TIMESTAMP	SQL92F	yes

Table1. Support for SQL-92 Features: Full character functions

Feature	Source	Derby
POSITION expression	SQL92F	use Java methods or LOCATE
UPPER/LOWER functions	SQL92F	yes

Table1. Support for SQL-92 Features: Miscellaneous

Feature	Source	Derby
Delimited identifiers	SQL92E	yes
Correlated subqueries	SQL92E	yes
Insert, Update, Delete statements	SQL92E	yes
Joins	SQL92E	yes
Where qualifications	SQL92E	yes
Group by	SQL92E	yes
Having	SQL92E	yes
Aggregate functions	SQL92E	yes
Order by	SQL92E	yes
Select expressions	SQL92E	yes
Select *	SQL92E	yes
SQLCODE	SQL92E	no, deprecated in SQL-92
SQLSTATE	SQL92E	yes
UNION, INTERSECT, and EXCEPT in views	SQL92T	yes
Implicit numeric casting	SQL92T	yes
Implicit character casting	SQL92T	yes
Get diagnostics	SQL92T	use JDBC *SQLExceptions*
Grouped operations	SQL92T	yes
Qualified * in select list	SQL92T	yes
Lowercase identifiers	SQL92T	yes
nullable PRIMARY KEYs	SQL92T	no
Multiple module support	SQL92T	no (not required and not part of JDBC)
Referential delete actions	SQL92T	CASCADE, SET NULL, RESTRICT, and NO ACTION.
CAST functions	SQL92T	yes
INSERT expressions	SQL92T	yes
Explicit defaults	SQL92T	yes
Keyword relaxations	SQL92T	yes

Feature	Source	Derby
Domain definition	SQL92I	no
CASE expression	SQL92I	partial support
Compound character string constants	SQL92I	use concatenation
LIKE enhancements	SQL92I	yes
UNIQUE predicate	SQL92I	no
Usage tables	SQL92I	*SYS.SYSDEPENDS*
Intermediate information schema	SQL92I	use JDBC *DatabaseMetaData* and Derby system tables
Subprogram support	SQL92I	not relevant to JDBC, which is much richer
Intermediate SQL Flagging	SQL92I	no
Schema manipulation	SQL92I	yes
Long identifiers	SQL92I	yes
Full outer join	SQL92I	no
Time zone specification	SQL92I	no
Scrolled cursors	SQL92I	partial (scrolling insensitive result sets through JDBC 2.0)
Intermediate set function support	SQL92I	partial
Character set definition	SQL92I	supports Java locales
Named character sets	SQL92I	supports Java locales
Scalar subquery values	SQL92I	yes
Expanded null predicate	SQL92I	yes
Constraint management	SQL92I	yes (ADD/DROP CONSTRAINT)
FOR BIT DATA types	SQL92F	yes
Assertion constraints	SQL92F	no
Temporary tables	SQL92F	IBM specific syntax only
Full dynamic SQL	SQL92F	no
Full value expressions	SQL92F	yes
Truth value tests	SQL92F	yes
Derived tables in FROM	SQL92F	yes
Trailing underscore	SQL92F	yes
Indicator data types	SQL92F	not relevant to JDBC
Referential name order	SQL92F	no
Full SQL Flagging	SQL92F	no
Row and table constructors	SQL92F	yes
Catalog name qualifiers	SQL92F	no
Simple tables	SQL92F	no
Subqueries in CHECK	SQL92F	no, but can do with Java methods
Union join	SQL92F	no
Collation and translation	SQL92F	Java locales supported

Feature	Source	Derby
Referential update actions	SQL92F	RESTRICT and NO ACTION. Can do others with triggers.
ALTER domain	SQL92F	no
INSERT column privileges	SQL92F	no
Referential MATCH types	SQL92F	no
View CHECK enhancements	SQL92F	no, views not updateable
Session management	SQL92F	use JDBC
Connection management	SQL92F	use JDBC
Self-referencing operations	SQL92F	yes
Insensitive cursors	SQL92F	Yes through JDBC 2.0
Full set function	SQL92F	partially
Catalog flagging	SQL92F	no
Local table references	SQL92F	no
Full cursor update	SQL92F	no

Derby System Tables

Derby includes system tables.

You can query system tables, but you cannot alter them.

All of the above 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.

The recommended way to get more information about these tables is to use an instance of the Java interface *java.sql.DatabaseMetaData*.

SYSALIASES

Describes the procedures and functions in the database.

Column Name	Type	Length	Nullability	Contents
ALIASID	CHAR	36	false	unique identifier for the alias
ALIAS	VARCHAR	128	false	alias
SCHEMAID	CHAR	36	true	reserved for future use
JAVACLASSNAM	LONGVARCHAR	255	false	the Java class name
ALIASTYPE	CHAR	1	false	*'F'* (function)*'P'* (procedure)
NAMESPACE	CHAR	1	false	*'F'* (function)*'P'* (procedure)
SYSTEMALIAS	BOOLEAN	'	false	*true* (system supplied or built-in alias) *false* (alias created by a user)
ALIASINFO	org.apache.derby.catalog.AliasInfo: This class is not part of the public API	'	true	A Java interface that encapsulates the additional information that is specific to an alias
SPECIFICNAME	VARCHAR	128	false	system-generated identifier

SYSCHECKS

Describes the check constraints within the current database.

Column Name	Type	Length	Nullability	Contents
CONSTRAINTID	CHAR	36	false	unique identifier for the constraint
CHECKDEFINITION	LONG VARCHAR	'	false	text of check constraint definition
REFERENCEDCOLUMNS	org.apache.derby.catalo ReferencedColumns:	'	false	description of the columns referenced by the check

Column Name	Type	Length	Nullability	Contents
	This class is not part of the public API.			constraint

SYSCOLUMNS

Describes the columns within all tables in the current database:

Column Name	Type	Length	Nullable	Contents
REFERENCEID	CHAR	36	false	Identifier for table (join with *SYSTABLES.TABLEID*)
COLUMNNAME	CHAR	128	false	column or parameter name
COLUMNNUMBER	INT	4	false	the position of the column within the table
COLUMNDATATYPE	org.apache.derby.catalo TypeDescriptor This class is not part of the public API.	'	false	system type that describes precision, length, scale, nullability, type name, and storage type of data
COLUMNDEFAULT	*java.io.Serializable*	'	true	for tables, describes default value of the column. The *toString()* method on the object stored in the table returns the text of the default value as specified in the CREATE TABLE or ALTER TABLE statement.
COLUMNDEFAULTID	CHAR	36	true	unique identifier for the default value
AUTOINCREMENT COLUMNVALUE	BIGINT	'	true	what the next value for column will be, if the column is an identity column
AUTOINCREMENT COLUMNSTART	BIGINT	'	true	initial value of column (if specified), if it is an identity column
AUTOINCREMENT COLUMNINC	BIGINT	'	true	amount column value is automatically incremented (if specified), if the column is an identity column

SYSCONGLOMERATES

Describes the conglomerates within the current database. A conglomerate is a unit of

storage and is either a table or an index.

Column Name	Type	Length	Nullable	Contents
SCHEMAID	CHAR	36	false	schema id for the conglomerate
TABLEID	CHAR	36	false	identifier for table (join with *SYSTABLES.TABLEID*)
CONGLOMERATENUMBER	BIGINT	8	false	conglomerate id for the conglomerate (heap or index)
CONGLOMERATENAME	VARCHAR	128	true	index name, if conglomerate is an index, otherwise the table ID
ISINDEX	BOOLEAN	1	false	whether or not conglomerate is an index
DESCRIPTOR	org.apache.derby.catalog.IndexDescriptc This class is not part of the public API.	'	true	system type describing the index
ISCONSTRAINT	BOOLEAN	1	true	whether or not conglomerate is a system-generated index enforcing a constraint
CONGLOMERATEID	CHAR	36	false	unique identifier for the conglomerate

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).

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	false	unique identifier for constraint
TABLEID	CHAR	36	false	identifier for table (join with *SYSTABLES.TABLEID*)
CONSTRAINTNAME	VARCHAR	128	false	constraint name (internally generated if not specified by user)
TYPE	CHAR	1	false	*P* (primary key), *U* (unique), *C* (check), or *F* (foreign key)
SCHEMAID	CHAR	36	false	identifier for schema that the constraint belongs to (join with *SYSSCHEMAS.SCHEMAID*)
STATE	CHAR	1	false	*E* for enabled, *D* for disabled
REFERENCECOUNT	INTEGER	1	false	the count of the number of foreign key constraints that reference this

Column Name	Type	Length	Nullable	Contents
				constraint; this number can be greater than zero only for PRIMARY KEY and UNIQUE constraints

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).

Providers are tables, conglomerates, and constraints. Dependents are views.

Column Name	Type	Length	Nullable	Contents
DEPENDENTID	CHAR	36	false	unique identifier for the dependent
DEPENDENTFINDER	org.apache.derby.catalo DependableFinder: This class is not part of the public API.	1	false	system type describing the view
PROVIDERID	CHAR	36	false	unique identifier for the provider
PROVIDERFINDER	org.apache.derby.catalo DependableFinder This class is not part of the public API.	1	false	system type describing the tables, conglomerates, and constraints that are providers

SYSFILES

Describes jar files stored in the database.

Column Name	Type	Length	Nullability	Contents
FILEID	CHAR	36	false	unique identifier for the jar file
SCHEMAID	CHAR	36	false	ID of the jar file's schema (join with *SYSSCHEMAS. SCHEMAID*)
FILENAME	VARCHAR	128	false	SQL name of the jar file
GENERATIONID	BIGINT	'	false	Generation number for the file. When jar files are replaced, their generation identifiers are changed.

SYSFOREIGNKEYS

Describes the information specific to foreign key constraints in the current database.

Derby generates a backing index for each foreign key constraint; the name of this index is the same as SYSFOREIGNKEYS.CONGLOMERATEID.

Column Name	Type	Length	Nullability	Contents
CONSTRAINTID	CHAR	36	false	unique identifier for the foreign key constraint (join with *SYSCONSTRAINTS. CONSTRAINTID*)
CONGLOMERATEID	CHAR	36	false	unique identifier for index backing up the foreign key constraint (join with *SYSCONGLOMERATES. CONGLOMERATEID*)
KEYCONSTRAINTID	CHAR	36	false	unique identifier for the primary key or unique constraint referenced by this foreign key (*SYSKEYS.CONSTRAINTID* or *SYSCONSTRAINTS. CONSTRAINTID*)
DELETERULE	CHAR	1	false	*R* for NO ACTION (default), *S* for RESTRICT, *C* for CASCADE, *U* for SET NULL
UPDATERULE	CHAR	1	false	*R for NO ACTION(default), S* for restrict

SYSKEYS

Describes the specific information for primary key and unique constraints within the current database. Derby generates an index on the table to back up each such constraint. The index name is the same as *SYSKEYS.CONGLOMERATEID*.

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	false	unique identifier for constraint
CONGLOMERATEID	CHAR	36	false	unique identifier for backing index

SYSSCHEMAS

Describes the schemas within the current database.

Column Name	Type	Length	Nullability	Contents
SCHEMAID	CHAR	36	false	unique identifier for the schema
SCHEMANAME	VARCHAR	128	false	schema name
AUTHORIZATIONID	VARCHAR	128	false	the authorization identifier of the owner of the schema

SYSSTATISTICS

Describes the schemas within the current database.

Column Name	Type	Length	Nullability	Contents
STATID	CHAR	36	false	unique identifier for the statistic
REFERENCEID	CHAR	36	false	the conglomerate for which the statistic was created (join with SYSCONGLOMERATES. CONGLOMERATEID)
TABLEID	CHAR	36	false	the table for which the information is collected
CREATIONTIMESTAMP	TIMESTAMP	'	false	time when this statistic was created or updated
TYPE	CHAR	1	false	type of statistics
VALID	BOOLEAN	'	false	whether the statistic is still valid
COLCOUNT	INTEGER	'	false	number of columns in the statistic
STATISTICS	org.apache. derby.catalog. Statistics: This class is not part of the public API.	'	true	statistics information

SYSSTATEMENTS

Contains one row per stored prepared statement.

Column Name	Type	Length	Nullability	Contents
STMTID	CHAR	36	false	unique identifier for the statement
STMTNAME	VARCHAR	128	false	name of the statement
SCHEMAID	CHAR	36	false	the schema in which the statement resides
TYPE	CHAR	1	false	always 'S'
VALID	BOOLEAN	'	false	TRUE if valid, FALSE if invalid
TEXT	LONG VARCHAR	'	false	text of the statement
LASTCOMPILED	TIMESTAMP	'	true	time that the statement was compiled
COMPILATION SCHEMAID	CHAR	36	false	id of the schema containing the statement
USINGTEXT	LONG VARCHAR	'	true	text of the USING clause of the CREATE STATEMENT and ALTER STATEMENT statements

SYSTABLES

Describes the tables and views within the current database.

Column Name	Type	Length	Nullable	Contents
TABLEID	CHAR	36	false	unique identifier for table or view
TABLENAME	VARCHAR	128	false	table or view name
TABLETYPE	CHAR	1	false	*'S'* (system table), '*T*' (user table), or '*V*' (view)
SCHEMAID	CHAR	36	false	schema id for the table or view
LOCK GRANULARITY	CHAR	1	false	Indicates the lock granularity for the table **'*T*'** (table level locking) **'*R*'** (row level locking, the default)

SYSTRIGGERS

Describes the database's triggers.

Column Name	Type	Length	Nullabilit	Contents
TRIGGERID	CHAR	36	false	unique identifier for the trigger
TRIGGERNAME	VARCHAR	128	false	name of the trigger
SCHEMAID	CHAR	36	false	id of the trigger's schema (join with *SYSSCHEMAS. SCHEMAID*)
CREATIONTIMESTAMP	TIMESTAMP	'	false	time the trigger was created
EVENT	CHAR	1	false	'U' for update, 'D' for delete, 'I' for insert
FIRINGTIME	CHAR	1	false	'B' for before 'A' for after
TYPE	CHAR	1	false	'R' for row, 'S' for statement
STATE	CHAR	1	false	'E' for enabled, 'D' for disabled
TABLEID	CHAR	36	false	id of the table on which the trigger is defined
WHENSTMTID	CHAR	36	true	used only if there is a WHEN clause (not yet supported)
ACTIONSTMTID	CHAR	36	true	id of the stored prepared statement for the triggered-SQL-statement (join with *SYSSTATEMENTS. STMTID*)
REFERENCEDCOLUMNS	org.apache.derby.catal ReferencedColumns: This class is not part of the public API.	'	true	descriptor of the columns referenced by UPDATE triggers
TRIGGERDEFINITION	LONG VARCHAR	'	true	text of the action SQL statement
REFERENCINGOLD	BOOLEAN	'	true	whether or not the

Column Name	Type	Length	Nullability	Contents
				OLDREFERENCINGNAME, if non-null, refers to the OLD row or table
REFERENCINGNEW	BOOLEAN	'	true	whether or not the NEWREFERENCINGNAME, if non-null, refers to the NEW row or table
OLDREFERENCINGNAME	VARCHAR	128	true	pseudoname as set using the REFERENCING OLD AS clause
NEWREFERENCINGNAME	VARCHAR	128	true	pseudoname as set using the REFERENCING NEW AS clause

Any SQL text that is part of a triggered-SQL-statement is compiled and stored in *SYSSTATEMENTS*. *ACTIONSTMTID* and *WHENSTMTID* are foreign keys that reference *SYSSTATEMENTS.STMTID*. The statements for a trigger are always in the same schema as the trigger.

SYSVIEWS

Describes the view definitions within the current database.

Column Name	Type	Length	Nullability	Contents
TABLEID	CHAR	36	false	unique identifier for the view (called TABLEID since it is joined with column of that name in SYSTABLES)
VIEWDEFINITION	LONG VARCHAR	'	false	text of view definition
CHECKOPTION	CHAR	1	false	'*N*' (check option not supported yet)
COMPILATION SCHEMAID	CHAR	36	false	id of the schema containing the view

Derby exception messages and SQL states

The JDBC driver returns *SQLException*s for all errors from Derby. If the exception originated in a user type but is not itself an *SQLException*, it is wrapped in an *SQLException*. Derby-specific *SQLException*s use *SQLState* class codes starting with *X*. Standard *SQLState* values are returned for exceptions where appropriate.

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 Derby.

Derby supplies values for the message and *SQLState* fields. In addition, Derby 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 Derby.

For information on processing *SQLExceptions*, see the *Derby Developer's Guide* .

SQLState and error message reference

The following tables list *SQLStates* for exceptions. Exceptions that begin with an *X* are specific to Derby. Note that some SQLStates specific to the network client might change in future releases.

Table1. Class Code 01: Warning

SQLSTATE	Message Text
01003	Null values were eliminated from the argument of a column function.
0100E	XX Attempt to return too many result sets.
01500	The constraint <*constraintName*> on table <*tableName*> has been dropped.
01501	The view <*viewName*> has been dropped.
01502	The trigger <*triggerName*> on table <*tableName*> has been dropped.
01503	The column <*columnName*> on table <*tableName*> has been modified by adding a not null constraint.
01504	The new index is a duplicate of an existing index: <*index*>.
01505	The value <*valueName*> may be truncated.
01522	The newly defined synonym '<*synonymName*>' resolved to the object '<*objectName*>' which is currently undefined.
01J01	Database '<*databaseName*>' not created, connection made to existing database instead.
01J02	Scroll sensitive cursors are not currently implemented.
01J03	Scroll sensitive and scroll insensitive updatable ResultSets are not currently implemented.
01J04	The class '<*className*>' for column '<*columnName*>' does not implement java.io.Serializable or java.sql.SQLData. Instances must implement one of these interfaces to allow them to be stored.
01J05	Database upgrade succeeded. The upgraded database is now ready for use. Revalidating stored prepared statements failed. See next exception for details about failure.

SQLSTATE	Message Text
01J06	ResultSet not updatable. Query does not qualify to generate an updatable ResultSet.

Table1. Class Code 04: Database authentication

SQLSTATE	Message Text
04501	Database connection refused.

Table1. Class Code 07: Dynamic SQL Error

SQLSTATE	Message Text
07000	At least one parameter to the current statement is uninitialized.
07004	Parameter <*parameterName*> is a <*procedureName*> procedure parameter and must be registered with CallableStatement.registerOutParameter before execution.
07009	No input parameters.

Table1. Class Code 08: Connection Exception

SQLSTATE	Message Text
08000	Connection closed by unknown interrupt.
08003	No current connection.
08004	Connection refused : <*connectionName*>
08006	Database '<*databaseName*>'

Table1. Class Code 0A: Feature Not Supported

SQLSTATE	Message Text
0A000	Feature not implemented: <*featureName*>.

Table1. Class Code 21: Cardinality Violation

SQLSTATE	Message Text
21000	Scalar subquery is only allowed to return a single row.

Table1. Class Code 22: Data Exception

SQLSTATE	Message Text
2200L	XMLPARSE operand is not an XML document; see next exception for details.
22001	A truncation error was encountered trying to shrink <*value*> '<*value*>' to length <*value*>
22003	The resulting value is outside the range for the data type <*datatypeName*>.
22005	An attempt was made to get a data value of type '<*typeName*>' from a data value of type '<*typeName*>'.
22007	The string representation of a datetime value is out of range.
22007	The syntax of the string representation of a datetime value is incorrect.
22008	'<*argument*>' is an invalid argument to the <*functionName*> function.
22011	The second or third argument of the SUBSTR function is out of range.
22012	Attempt to divide by zero.

SQLSTATE	Message Text
22013	Attempt to take the square root of a negative number, '<*number*>'.
22014	The start position for LOCATE is invalid; it must be a positive integer. The index to start the search from is '<*index*>'. The string to search for is '<*index*>'. The string to search from is '<*index*>'.
22015	The '<*functionName*>' function is not allowed on the following set of types. First operand is of type '<*typeName*>'. Second operand is of type '<*typeName*>'. Third operand (start position) is of type '<*typeName*>'.
22018	Invalid character string format for type <*typeName*>.
22019	Invalid escape sequence, '<*sequenceName*>'. The escape string must be exactly one character. It cannot be a null or more than one character.
22025	Escape character must be followed by escape character, _, or %. It cannot be followed by any other character or be at the end of the pattern.
22027	The built-in TRIM() function only supports a single trim character. The LTRIM() and RTRIM() built-in functions support multiple trim characters.
22501	An ESCAPE clause of NULL returns undefined results and is not allowed.

Table1. Class Code 23: Constraint Violation

SQLSTATE	Message Text
23502	Column '<*columnName*>' cannot accept a NULL value.
23503	<*value*> on table '<*tableName*>' caused a violation of foreign key constraint '<*constraintName*>' for key <*keyName*>. The statement has been rolled back.
23505	The statement was aborted because it would have caused a duplicate key value in a unique or primary key constraint or unique index identified by '<*value*>' defined on '<*value*>'.
23513	The check constraint '<*constraintName*>' was violated while performing an INSERT or UPDATE on table '<*tableName*>'.

Table1. Class Code 24: Invalid Cursor State

SQLSTATE	Message Text
24000	Invalid cursor state - no current row.

Table1. Class Code 25: Invalid Transaction State

SQLSTATE	Message Text
25000	Invalid transaction state.
25501	Unable to set the connection read-only property in an active transaction.
25502	An SQL data change is not permitted for a read-only connection, user or database.
25503	DDL is not permitted for a read-only connection, user or database.
25505	A read-only user or a user in a read-only database is not permitted to disable read-only mode on a connection.

Table1. Class Code 28: Invalid Authorization Specification

SQLSTATE	Message Text
28501	Invalid database authorization property '<*value*>=<*value*>'.

SQLSTATE	Message Text
28502	The user name '<*userName*>' is not valid.
28503	User(s) '<*userName*>' must not be in both read-only and full-access authorization lists.
28504	Repeated user(s) '<*userName*>' in access list '<*listName*>'.

Table1. Class Code 38: External Function Exception

SQLSTATE	Message Text
38000	The exception '<*exception*>' was thrown while evaluating an expression.
38001	The external routine is not allowed to execute SQL statements.
38002	The routine attempted to modify data, but the routine was not defined as MODIFIES SQL DATA.
38004	The routine attempted to read data, but the routine was not defined as READS SQL DATA.

Table1. Class Code 39: External Function Exception

SQLSTATE	Message Text
39004	A NULL value cannot be passed to a method which takes a parameter of primitive type '<*type*>'.

Table1. Class Code 3B: Invalid SAVEPOINT

SQLSTATE	Message Text
3B001	SAVEPOINT, <*savepoint*> does not exist or is not active in the current transaction.
3B002	The maximum number of savepoints has been reached.
3B501	A SAVEPOINT with the passed name already exists in the current transaction.
3B502	A RELEASE or ROLLBACK TO SAVEPOINT was specified, but the savepoint does not exist.

Table1. Class Code 40: Transaction Rollback

SQLSTATE	Message Text
40001	A lock could not be obtained due to a deadlock, cycle of locks and waiters is: <*value*>. The selected victim is XID : <*value*>.
40XC0	Dead statement. This may be caused by catching a transaction severity error inside this statement.
40XD0	Container has been closed.
40XD1	Container was opened in read-only mode.
40XD2	Container <*containerName*> cannot be opened; it either has been dropped or does not exist.
40XL1	A lock could not be obtained within the time requested.
40XL2	A lock could not be obtained within the time requested. The lockTable dump is: <*tableDump*>.
40XT0	An internal error was identified by RawStore module.
40XT1	An exception was thrown during transaction commit.
40XT2	An exception was thrown during rollback of a SAVEPOINT.

SQLSTATE	Message Text
40XT4	An attempt was made to close a transaction that was still active. The transaction has been aborted.
40XT5	Exception thrown during an internal transaction.
40XT6	Database is in quiescent state, cannot activate transaction. Please wait for a moment until it exits the inactive state.
40XT7	Operation is not supported in an internal transaction.

Table1. Class Code 42: Syntax Error or Access Rule Violation

SQLSTATE	Message Text
42000	Syntax error or access rule violation; see additional errors for details.
42601	ALTER TABLE statement cannot add an IDENTITY column to a table.
42601	In an ALTER TABLE statement, the column '<*columnName*>' has been specified as NOT NULL and either the DEFAULT clause was not specified or was specified as DEFAULT NULL.
42605	The number of arguments for function '<*functionName*>' is incorrect.
42606	An invalid hexadecimal constant starting with '<*number*>' has been detected.
42610	All the arguments to the COALESCE/VALUE function cannot be parameters. The function needs at least one argument that is not a parameter.
42611	The length, precision, or scale attribute for column, or type mapping '<*value*>' is not valid.
42613	Multiple or conflicting keywords involving the '<*clause*>' clause are present.
42621	A check constraint or generated column that is defined with '<*value*>' is invalid.
42622	The name '<*name*>' is too long. The maximum length is '<*maximumLength*>'.
42734	Name '<*name*>' specified in context '<*context*>' is not unique.
42802	The number of values assigned is not the same as the number of specified or implied columns.
42803	An expression containing the column '<*columnName*>' appears in the SELECT list and is not part of a GROUP BY clause.
42815	The replacement value for '<*value*>' is invalid.
42815	The data type, length or value of arguments '<*firstArguement*>' and '<*secondArguement*>' is incompatible.
42818	Comparisons between '<*value*>' and '<*value*>' are not supported.
42820	The floating point literal '<*string*>' contains more than 30 characters.
42821	Columns of type '<*type*>' cannot hold values of type '<*type*>'.
42824	An operand of LIKE is not a string, or the first operand is not a column.
42831	'<*columnName*>' cannot be a column of a primary key or unique key because it can contain null values.
42834	SET NULL cannot be specified because FOREIGN KEY '<*key*>' cannot contain null values.
42837	ALTER TABLE '<*tableName*>' specified attributes for column '<*columnName*>' that are not compatible with the existing column.
42846	Cannot convert types '<*type*>' to '<*type*>'.
42877	A qualified column name '<*columnName*>' is not allowed in the ORDER BY clause.
42884	No authorized routine named '<*routineName*>' of type '<*type*>' having compatible

SQLSTATE	Message Text
	arguments was found.
42886	'<*value*>' parameter '<*value*>' requires a parameter marker '<*parameter*>'.
42894	DEFAULT value or IDENTITY attribute value is not valid for column '<*columnName*>'.
428C1	Only one identity column is allowed in a table.
428EK	The qualifier for a declared global temporary table name must be SESSION.
42903	Invalid use of an aggregate function.
42908	The CREATE VIEW statement does not include a column list.
42915	Foreign Key '<*key*>' is invalid because '<*value*>'.
42916	Synonym '<*synonym2*>' cannot be created for '<*synonym1*>' as it would result in a circular synonym chain.
42939	An object cannot be created with the schema name '<*schemaName*>'.
42962	Long column type column or parameter '<*columnName*>' not permitted in declared global temporary tables or procedure definitions.
42972	An ON clause associated with a JOIN operator is not valid.
42995	The requested function does not apply to global temporary tables.
42X01	Syntax error: <*error*>.
42X02	<*value*>.
42X03	Column name '<*columnName*>' is in more than one table in the FROM list.
42X04	Column '<*columnName*>' is either not in any table in the FROM list or appears within a join specification and is outside the scope of the join specification or appears in a HAVING clause and is not in the GROUP BY list. If this is a CREATE or ALTER TABLE statement then '<*columnName*>' is not a column in the target table.
42X05	Table '<*tableName*>' does not exist.
42X06	Too many result columns specified for table '<*tableName*>'.
42X07	Null is only allowed in a VALUES clause within an INSERT statement.
42X08	The constructor for class '<*className*>' cannot be used as an external virtual table because the class does not implement '<*constructorName*>'.
42X09	The table or alias name '<*tableName*>' is used more than once in the FROM list.
42X10	'<*tableName*>' is not an exposed table name in the scope in which it appears.
42X12	Column name '<*tableName*>' appears more than once in the CREATE TABLE statement.
42X13	Column name '<*columnName*>' appears more than once in the column list of an INSERT statement.
42X14	'<*columnName*>' is not a column in table or VTI '<*value*>'.
42X15	Column name '<*columnName*>' appears in a statement without a FROM list.
42X16	Column name '<*columnName*>' appears multiple times in the SET clause of an UPDATE statement.
42X17	In the Properties list of a FROM clause, the value '<*value*>' is not valid as a joinOrder specification. Only the values FIXED and UNFIXED are valid.
42X19	The WHERE or HAVING clause or CHECK CONSTRAINT definition is a '<*value*>' expression. It must be a BOOLEAN expression.
42X23	Cursor <*cursorName*> is not updatable.
42X25	The '<*functionName*>' function is not allowed on the '<*type*>' type.
42X26	The class '<*className*> for column '<*columnName*>' does not exist or is inaccessible.

SQLSTATE	Message Text
	This can happen if the class is not public.
42X28	Delete table '<*tableName*>' is not target of cursor '<*cursorName*>'.
42X29	Update table '<*tableName*>' is not the target of cursor '<*cursorName*>'.
42X30	Cursor '<*cursorName*>' not found. Verify that autocommit is OFF.
42X31	Column '<*columnName*>' is not in the FOR UPDATE list of cursor '<*cursorName*>'.
42X32	The number of columns in the derived column list must match the number of columns in table '<*tableName*>'.
42X33	The derived column list contains a duplicate column name '<*columnName*>'.
42X34	There is a ? parameter in the select list. This is not allowed.
42X35	It is not allowed for both operands of '<*value*>' to be ? parameters.
42X36	The '<*operator*>' operator is not allowed to take a ? parameter as an operand.
42X37	The unary '<*operator*>' operator is not allowed on the '<*type*>' type.
42X38	'SELECT *' only allowed in EXISTS and NOT EXISTS subqueries.
42X39	Subquery is only allowed to return a single column.
42X40	A NOT statement has an operand that is not boolean . The operand of NOT must evaluate to TRUE, FALSE, or UNKNOWN.
42X41	In the Properties clause of a FROM list, the property '<*propertyName*>' is not valid (the property was being set to '<*value*>').
42X42	Correlation name not allowed for column '<*columnName*>' because it is part of the FOR UPDATE list.
42X43	The ResultSetMetaData returned for the class/object '<*className*>' was null. In order to use this class as an external virtual table, the ResultSetMetaData cannot be null.
42X44	Invalid length '<*length*>' in column specification.
42X45	<*type*> is an invalid type for argument number <*value*> of <*value*>.
42X48	Value '<*value*>' is not a valid precision for <*value*>.
42X49	Value '<*value*>' is not a valid integer literal.
42X50	No method was found that matched the method call <*methodName*>.<*value*>(<*value*>), tried all combinations of object and primitive types and any possible type conversion for any parameters the method call may have. The method might exist but it is not public and/or static, or the parameter types are not method invocation convertible.
42X51	The class '<*className*>' does not exist or is inaccessible. This can happen if the class is not public.
42X52	Calling method ('<*methodName*>') using a receiver of the Java primitive type '<*type*>" is not allowed.
42X53	The LIKE predicate can only have 'CHAR' or 'VARCHAR' operands. Type '<*type*>' is not permitted.
42X54	The Java method '<*methodName*>' has a ? as a receiver. This is not allowed.
42X55	Table name '<*tableName*>' should be the same as '<*value*>'.
42X56	The number of columns in the view column list does not match the number of columns in the underlying query expression in the view definition for '<*value*>'.
42X57	The getColumnCount() for external virtual table '<*tableName*>' returned an invalid value '<*value*>'. Valid values are greater than or equal to 1.
42X58	The number of columns on the left and right sides of the <*tableName*> must be the same.

SQLSTATE	Message Text
42X59	The number of columns in each VALUES constructor must be the same.
42X60	Invalid value '<*value*>' for insertMode property specified for table '<*tableName*>'.
42X61	Types '<*type*>' and '<*type*>' are not <*value*> compatible.
42X62	'<*value*>' is not allowed in the '<*schemaName*>' schema.
42X63	The USING clause did not return any results. No parameters can be set.
42X64	In the Properties list, the invalid value '<*value*>' was specified for the useStatistics property. The only valid values are TRUE or FALSE .
42X65	Index '<*index*>' does not exist.
42X66	Column name '<*columnName*>' appears more than once in the CREATE INDEX statement.
42X68	No field '<*fieldName*>' was found belonging to class '<*className*>'. The field might exist, but it is not public, or the class does not exist or is not public.
42X69	It is not allowed to reference a field ('<*fieldName*>') using a referencing expression of the Java primitive type '<*type*>'.
42X72	No static field '<*fieldName*>' was found belonging to class '<*className*>'. The field might exist, but it is not public and/or static, or the class does not exist or the class is not public.
42X73	Method resolution for signature <*value*>.<*value*>(<*value*>) was ambiguous. (No single maximally specific method.)
42X74	Invalid CALL statement syntax.
42X75	No constructor was found with the signature <*value*>(<*value*>). It may be that the parameter types are not method invocation convertible.
42X76	At least one column, '<*columnName*>', in the primary key being added is nullable. All columns in a primary key must be non-nullable.
42X77	Column position '<*columnPosition*>' is out of range for the query expression.
42X78	Column '<*columnName*>' is not in the result of the query expression.
42X79	Column name '<*columnName*>' appears more than once in the result of the query expression.
42X80	VALUES clause must contain at least one element. Empty elements are not allowed.
42X82	The USING clause returned more than one row. Only single-row ResultSets are permissible.
42X83	The constraints on column '<*columnName*>' require that it be both nullable and not nullable.
42X84	Index '<*index*>' was created to enforce constraint '<*constraintName*>'. It can only be dropped by dropping the constraint.
42X85	Constraint '<*constraintName*>' is required to be in the same schema as table '<*tableName*>'.
42X86	ALTER TABLE failed. There is no constraint '<*constraintName*>' on table '<*tableName*>'.
42X87	At least one result expression (THEN or ELSE) of the '<*expression*>' expression must not be a '?'.
42X88	A conditional has a non-boolean operand. The operand of a conditional must evaluate to TRUE, FALSE, or UNKNOWN.
42X89	Types '<*type*>' and '<*type*>' are not type compatible. Neither type is assignable to the other type.
42X90	More than one primary key constraint specified for table '<*tableName*>'.

SQLSTATE	Message Text
42X91	Constraint name '<*constraintName*>' appears more than once in the CREATE TABLE statement.
42X92	Column name '<*columnName*>' appears more than once in a constraint's column list.
42X93	Table '<*tableName*>' contains a constraint definition with column '<*columnName*>' which is not in the table.
42X94	<*value*> '<*value*>' does not exist.
42X96	The database classpath contains an unknown jar file '<*fileName*>'.
42X98	Parameters are not allowed in a VIEW definition.
42Y00	Class '<*className*>' does not implement org.apache.derby.iapi.db.AggregateDefinition and thus cannot be used as an aggregate expression.
42Y01	Constraint '<*constraintName*>' is invalid.
42Y03	'<*statement*>' is not recognized as a function or procedure.
42Y04	Cannot create a procedure or function with EXTERNAL NAME '<*name*>' because it is not a list separated by periods. The expected format is <*fulljavapath*>.<*method name*>.
42Y05	There is no Foreign Key named '<*key*>'.
42Y07	Schema '<*schemaName*>' does not exist.
42Y08	Foreign key constraints are not allowed on system tables.
42Y09	Void methods are only allowed within a CALL statement.
42Y10	A table constructor that is not in an INSERT statement has all ? parameters in one of its columns. For each column, at least one of the rows must have a non-parameter.
42Y11	A join specification is required with the '<*clauseName*>' clause.
42Y12	The ON clause of a JOIN is a '<*expressionType*>' expression. It must be a BOOLEAN expression.
42Y13	Column name '<*columnName*>' appears more than once in the CREATE VIEW statement.
42Y16	No public static method '<*methodName*>' was found in class '<*className*>'. The method might exist, but it is not public, or it is not static.
42Y19	'<*columnName*>' appears multiple times in the GROUP BY list. Columns in the GROUP BY list must be unambiguous.
42Y22	Aggregate <*aggregateType*> cannot operate on type <*type*>.
42Y23	Incorrect JDBC type info returned for column <*columnName*>.
42Y24	View '<*viewName*>' is not updatable. (Views are currently not updatable.)
42Y25	'<*tableName*>' is a system table. Users are not allowed to modify the contents of this table.
42Y27	Parameters are not allowed in the trigger action.
42Y29	The SELECT list of a non-grouped query contains at least one invalid expression. When the SELECT list contains at least one aggregate then all entries must be valid aggregate expressions.
42Y30	The SELECT list of a grouped query contains at least one invalid expression. If a SELECT list has a GROUP BY, the list may only contain grouping columns and valid aggregate expressions.
42Y32	Aggregator class '<*className*> aggregate '<*aggregateName*>' on type <*type*> does not implement com.ibm.db2j.aggregates.Aggregator.
42Y33	Aggregate <*aggregateName*> contains one or more aggregates.

SQLSTATE	Message Text
42Y34	Column name '<*columnName*>' matches more than one result column in table '<*tableName*>'.
42Y35	Column reference '<*reference*>' is invalid. When the SELECT list contains at least one aggregate then all entries must be valid aggregate expressions.
42Y36	Column reference '<*value*>' is invalid. For a SELECT list with a GROUP BY, the list may only contain grouping columns and valid aggregate expressions.
42Y37	'<*value*>' is a Java primitive and cannot be used with this operator.
42Y38	`insertMode = replace` is not permitted on an insert where the target table, '<*tableName*>', is referenced in the SELECT.
42Y39	'<*value*>' may not appear in a CHECK CONSTRAINT definition because it may return non-deterministic results.
42Y40	'<*value*>' appears multiple times in the UPDATE OF column list for trigger '<*triggerName*>'.
42Y41	'<*value*>' cannot be directly invoked via EXECUTE STATEMENT because it is part of a trigger.
42Y42	Scale '<*scaleValue*>' is not a valid scale for a <*value*>.
42Y43	Scale '<*scaleValue*>' is not a valid scale with precision of '<*precision*>'.
42Y44	Invalid key '<*key*>' specified in the Properties list of a FROM list. The case-sensitive keys that are currently supported are '<*key*>'.
42Y45	VTI '<*value*>' cannot be bound because it is a special trigger VTI and this statement is not part of a trigger action or WHEN clause.
42Y46	Invalid Properties list in FROM list. There is no index '<*index*>' on table '<*tableName*>'.
42Y48	Invalid Properties list in FROM list. Either there is no named constraint '<*constraintName*>' on table '<*tableName*>' or the constraint does not have a backing index.
42Y49	Multiple values specified for property key '<*key*>'.
42Y50	Properties list for table '<*tableName*>' may contain values for index or for constraint but not both.
42Y55	'<*value*>' cannot be performed on '<*value*>' because it does not exist.
42Y56	Invalid join strategy '<*strategyValue*>' specified in Properties list on table '<*tableName*>'. The currently supported values for a join strategy are: `hash` and `nestedloop` .
42Y58	NumberFormatException occurred when converting value '<*value*>' for optimizer override '<*value*>'.
42Y59	Invalid value, '<*value*>', specified for hashInitialCapacity override. Value must be greater than 0.
42Y60	Invalid value, '<*value*>', specified for hashLoadFactor override. Value must be greater than 0.0 and less than or equal to 1.0.
42Y61	Invalid value, '<*value*>' specified for hashMaxCapacity override. Value must be greater than 0.

SQLSTATE	Message Text
42Y62	'<*value*>' is not allowed on '<*value*>' because it is a view.
42Y63	Hash join requires an optimizable equijoin predicate on a column in the selected index or heap. An optimizable equijoin predicate does not exist on any column in table or index '<*index*>'. Use the 'index' optimizer override to specify such an index or the heap on table '<*tableName*>'.
42Y64	bulkFetch value of '<*value*>' is invalid. The minimum value for bulkFetch is 1.
42Y65	bulkFetch is not permitted on '<*joinType*>' joins.
42Y66	bulkFetch is not permitted on updatable cursors.
42Y67	Schema '<*schemaName*>' cannot be dropped.
42Y69	No valid execution plan was found for this statement. This may have one of two causes: either you specified a hash join strategy when hash join is not allowed (no optimizable equijoin) or you are attempting to join two external virtual tables, each of which references the other, and so the statement cannot be evaluated.
42Y70	The user specified an illegal join order. This could be caused by a join column from an inner table being passed as a parameter to an external virtual table.
42Y71	System function or procedure '<*procedureName*>' cannot be dropped.
42Y82	System generated stored prepared statement '<*statement*>' that cannot be dropped using DROP STATEMENT. It is part of a trigger.
42Y83	An untyped null is not permitted as an argument to aggregate <*aggregateName*>. Please cast the null to a suitable type.
42Y84	'<*value*>' may not appear in a DEFAULT definition.
42Y85	The DEFAULT keyword is only allowed in a VALUES clause when the VALUES clause appears within an INSERT statement.
42Y90	FOR UPDATE is not permitted in this type of statement.
42Y91	The USING clause is not permitted in an EXECUTE STATEMENT for a trigger action.
42Y92	<*triggerName*> triggers may only reference <*value*> transition variables/tables.
42Y93	Illegal REFERENCING clause: only one name is permitted for each type of transition variable/table.
42Y94	An AND or OR has a non-boolean operand. The operands of AND and OR must evaluate to TRUE, FALSE, or UNKNOWN.
42Y95	The '<*operatorName*>' operator with a left operand type of '<*operandType*>' and a right operand type of '<*operandType*>' is not supported.
42Y97	Invalid escape character at line '<*lineNumber*>', column '<*columnName*>'.
42Z02	Multiple DISTINCT aggregates are not supported at this time.
42Z07	Aggregates are not permitted in the ON clause.
42Z08	Bulk insert replace is not permitted on '<*value*>' because it has an enabled trigger (<*value*>).
42Z15	Invalid type specified for column '<*columnName*>'. The type of a column may not be changed.
42Z16	Only columns of type VARCHAR may have their length altered.
42Z17	Invalid length specified for column '<*columnName*>'. Length must be greater than the current column length.
42Z18	Column '<*columnName*>' is part of a foreign key constraint '<*constraintName*>'. To alter the length of this column, you should drop the constraint first, perform the ALTER TABLE, and then recreate the constraint.
42Z19	Column '<*columnName*>' is being referenced by at least one foreign key constraint

SQLSTATE	Message Text
	constraintName>'. To alter the length of this column, you should drop referencing constraints, perform the ALTER TABLE, and then recreate the constraints.
42Z20	Column '<*columnName*>' cannot be made nullable. It is part of a primary key, which cannot have any nullable columns.
42Z21	Invalid increment specified for identity for column '<*columnName*>'. Increment cannot be zero.
42Z22	Invalid type specified for identity column '<*columnName*>'. The only valid types for identity columns are BIGINT, INT and SMALLINT.
42Z23	Attempt to modify an identity column '<*columnName*>'.
42Z24	Overflow occurred in identity value for column '<*columnName*>' in table '<*tableName*>'.
42Z25	INTERNAL ERROR identity counter. Update was called without arguments with `current value = NULL` .
42Z26	A column, '<*columnName*>', with an identity default cannot be made nullable.
42Z27	A nullable column, '<*columnName*>', cannot be modified to have identity default.
42Z50	INTERNAL ERROR: Unable to generate code for <*value*>.
42Z53	INTERNAL ERROR: Do not know what type of activation to generate for node choice <*value*>.
42Z90	Class '<*className*>' does not return an updatable ResultSet.
42Z91	subquery
42Z92	repeatable read
42Z93	Constraints '<*constraintName*>' and '<*constraintName*>' have the same set of columns, which is not allowed.
42Z97	Renaming column '<*columnName*>' will cause check constraint '<*constraintName*>' to break.
42Z99	String or Hex literal cannot exceed 64K.
42Z9A	read uncommitted
42Z9B	The external virtual table interface does not support BLOB or CLOB columns. '<*value*>' column '<*value*>'.
42Z9D	'<*statement*>' statements are not allowed in '<*triggerName*>' triggers.
42Z9E	Constraint '<*constraintName*>' is not a <*value*> constraint.
42Z9F	Too many indexes (<*index*>) on the table <*tableName*>. The limit is <*number*>.

Table1. Class Code XOX: Execution exceptions

SQLState	Message Text
X0X02	Table '<*tableName*>' cannot be locked in '<*mode*>' mode.
X0X03	Invalid transaction state - held cursor requires same isolation level
X0X05	Table '<*tableName*>' does not exist.
X0X07	Cannot drop jar file '<*fileName*>' because its on your db2j.database.classpath '<*classpath*>'.
X0X0E	The column position '<*columnPosition*>' listed in the auto-generated column selection array was not found in the insert table.
X0X0F	Column name '<*columnName*>' listed in auto-generated column selection array not

SQLState	Message Text
	found in the insert table.
X0X10	The USING clause returned more than one row; only single-row ResultSets are permissible.
X0X11	The USING clause did not return any results so no parameters can be set.
X0X13	Jar file '<*fileName*>' does not exist in schema '<*schemaName*>'.
X0X14	Binding directly to an XML value is not allowed; try using XMLPARSE.
X0X15	XML values are not allowed in top-level result sets; try using XMLSERIALIZE.
X0X16	XML syntax error; missing keyword(s): '<*keywords*>'.
X0X17	Invalid target type for XMLSERIALIZE: '<*value*>'.
X0X18	XML feature not supported: '<*featureName*>'.
X0X57	An attempt was made to put a Java value of type '<*type*>' into a SQL value, but there is no corresponding SQL type. The Java value is probably the result of a method call or field access.
X0X60	A cursor with name '<*cursorName*>' already exists.
X0X61	The values for column '<*value*>' in index '<*value*>' and table '<*value*>.<*value*>' do not match for row location <*value*>. The value in the index is '<*value*>', while the value in the base table is '<*value*>'. The full index key, including the row location, is '<*value*>'. The suggested corrective action is to recreate the index.
X0X62	Inconsistency found between table '<*tableName*>' and index '<*index*>'. Error when trying to retrieve row location '<*rowLocation*>' from the table. The full index key, including the row location, is '<*index*>'. The suggested corrective action is to recreate the index.
X0X63	Got IOException '<*value*>'.
X0X67	Columns of type '<*type*>' may not be used in CREATE INDEX, ORDER BY, GROUP BY, UNION, INTERSECT, EXCEPT or DISTINCT statements because comparisons are not supported for that type.
X0X81	<*value*> '<*value*>' does not exist.
X0X85	Index '<*index*>' was not created because '<*type*>' is not a valid index type.
X0X86	0 is an invalid parameter value for `ResultSet.absolute(int row)` .
X0X87	`ResultSet.relative(int row)` cannot be called when the cursor is not positioned on a row.
X0X95	Operation '<*operation*>' cannot be performed on object '<*object*>' because there is an open ResultSet dependent on that object.
X0X99	Index '<*index*>' does not exist.
X0XML	Encountered unexpected error while processing XML; see next exception for details.

Table1. Class Code X0Y: Execution exceptions

SQLSTATE	Message Text
X0Y16	'<*value*>' is not a view. If it is a table, then use DROP TABLE instead.
X0Y23	Operation '<*operation*>' cannot be performed on object '<*object*>' because VIEW '<*viewName*>' is dependent on that object.

SQLSTATE	Message Text
X0Y24	Operation '<*operation*>' cannot be performed on object '<*object*>' because STATEMENT '<*statement*>' is dependent on that object.
X0Y25	Operation '<*value*>' cannot be performed on object '<*value*>' because <*value*> '<*value*>' is dependent on that object.
X0Y26	Index '<*index*>' is required to be in the same schema as table '<*tableName*>'.
X0Y28	Index '<*index*>' cannot be created on system table '<*tableName*>'. Users cannot create indexes on system tables.
X0Y32	<*value*> '<*value*>' already exists in <*value*> '<*value*>'.
X0Y38	Cannot create index '<*index*>' because table '<*tableName*>' does not exist.
X0Y41	Constraint '<*constraintName*>' is invalid because the referenced table <*tableName*> has no primary key. Either add a primary key to <*tableName*> or explicitly specify the columns of a unique constraint that this foreign key references.
X0Y42	Constraint '<*constraintName*>' is invalid: the types of the foreign key columns do not match the types of the referenced columns
X0Y43	Constraint '<*value*>' is invalid: the number of columns in <*value*> (<*value*>) does not match the number of columns in the referenced key (<*value*>).
X0Y44	Constraint '<*constraintName*>' is invalid: there is no unique or primary key constraint on table '<*tableName*>' that matches the number and types of the columns in the foreign key.
X0Y45	Foreign key constraint '<*constraintName*>' cannot be added to or enabled on table <*tableName*> because one or more foreign keys do not have matching referenced keys.
X0Y46	Constraint '<*constraintName*>' is invalid: referenced table <*tableName*> does not exist.
X0Y54	Schema '<*schemaName*>' cannot be dropped because it is not empty.
X0Y55	The number of rows in the base table does not match the number of rows in at least 1 of the indexes on the table. Index '<*value*>' on table '<*value*>.<*value*>' has <*value*> rows, but the base table has <*value*> rows. The suggested corrective action is to recreate the index.
X0Y56	'<*value*>' is not allowed on the System table '<*value*>'.
X0Y57	A non-nullable column cannot be added to table '<*tableName*>' because the table contains at least one row. Non-nullable columns can only be added to empty tables.
X0Y58	Attempt to add a primary key constraint to table '<*tableName*>' failed because the table already has a constraint of that type. A table can only have a single primary key constraint.
X0Y59	Attempt to add or enable constraint(s) on table '<*tableName*>' failed because the table contains <*rowName*> row(s) that violate the following check constraint(s): <*constraintName*>.
X0Y63	The command on table '<*tableName*>' failed because null data was found in the primary key or unique constraint/index column(s). All columns in a primary or unique index key must not be null.
X0Y66	Cannot issue commit in a nested connection when there is a pending operation in the parent connection.
X0Y67	Cannot issue rollback in a nested connection when there is a pending operation in the parent connection.
X0Y68	<*value*> '<*value*>' already exists.
X0Y69	<*value*> is not permitted because trigger <*triggerName*> is active on <*value*>.
X0Y70	INSERT, UPDATE and DELETE are not permitted on table <*tableName*> because trigger <*triggerName*> is active.

SQLSTATE	Message Text
X0Y71	Transaction manipulation such as SET ISOLATION is not permitted because trigger <*triggerName*> is active.
X0Y72	Bulk insert replace is not permitted on '<*value*>' because it has an enabled trigger (<*value*>).
X0Y77	Cannot issue set transaction isolation statement on a global transaction that is in progress because it would have implicitly committed the global transaction.
X0Y78	Statement.executeQuery() cannot be called with a statement that returns a row count.
X0Y79	Statement.executeUpdate() cannot be called with a statement that returns a ResultSet.
X0Y80	ALTER table '<*tableName*>' failed. Null data found in column '<*columnName*>'.
X0Y83	WARNING: While deleting a row from a table the index row for base table row <*rowName*> was not found in index with conglomerate id <*id*>. This problem has automatically been corrected as part of the delete operation.

Table1. Class Code XBCA: CacheService

SQLSTATE	Message Text
XBCA0	Cannot create new object with key <*key*> in <*cache*> cache. The object already exists in the cache.

Table1. Class Code XBCM: ClassManager

SQLSTATE	Message Text
XBCM1	Java linkage error thrown during load of generated class <*className*>.
XBCM2	Cannot create an instance of generated class <*className*>.
XBCM3	Method <*methodName*> () does not exist in generated class <*className*>.

Table1. Class Code XBCX: Cryptography

SQLSTATE	Message Text
XBCX0	Exception from Cryptography provider. See next exception for details.
XBCX1	Initializing cipher with illegal mode, must be either ENCRYPT or DECRYPT.
XBCX2	Initializing cipher with a boot password that is too short. The password must be at least <*number*> characters long.
XBCX5	Cannot change boot password to null.
XBCX6	Cannot change boot password to a non-string serializable type.
XBCX7	Wrong format for changing boot password. Format must be : `old_boot_password, new_boot_password`.
XBCX8	Cannot change boot password for a non-encrypted database.
XBCX9	Cannot change boot password for a read-only database.
XBCXA	Wrong boot password.
XBCXB	Bad encryption padding '<*value*>' or padding not specified. 'NoPadding' must be used.
XBCXC	Encryption algorithm '<*algorithmName*>' does not exist. Please check that the chosen provider '<*providerName*>' supports this algorithm.
XBCXD	The encryption algorithm cannot be changed after the database is created.
XBCXE	The encryption provider cannot be changed after the database is created.

SQLSTATE	Message Text
XBCXF	The class '<*className*>' representing the encryption provider cannot be found.
XBCXG	The encryption provider '<*providerName*>' does not exist.
XBCXH	The encryptionAlgorithm '<*algorithmName*>' is not in the correct format. The correct format is `algorithm/feedbackMode/NoPadding`.
XBCXI	The feedback mode '<*mode*>' is not supported. Supported feedback modes are CBC, CFB, OFB and ECB.
XBCXJ	The application is using a version of the Java Cryptography Extension (JCE) earlier than 1.2.1. Please upgrade to JCE 1.2.1 and try the operation again.
XBCXK	The given encryption key does not match the encryption key used when creating the database. Please ensure that you are using the correct encryption key and try again.
XBCXL	The verification process for the encryption key was not successful. This could have been caused by an error when accessing the appropriate file to do the verification process. See next exception for details.

Table1. Class Code XBM: Monitor

SQLSTATE	Message Text
XBM01	Startup failed due to an exception. See next exception for details.
XBM02	Startup failed due to missing functionality for <*value*>. Please ensure your classpath includes the correct Derby Derby software.
XBM05	Startup failed due to missing product version information for <*value*>.
XBM06	Startup failed. An encrypted database cannot be accessed without the correct boot password.
XBM07	Startup failed. Boot password must be at least 8 bytes long.
XBM08	Could not instantiate <*value*> StorageFactory class <*value*>.
XBM0G	Failed to start encryption engine. Please make sure you are running Java 2 and have downloaded an encryption provider such as jce and put it in your classpath.
XBM0H	Directory <*directoryName*> cannot be created.
XBM0I	Directory <*directoryName*> cannot be removed.
XBM0J	Directory <*directoryName*> already exists.
XBM0K	Unknown sub-protocol for database name <*databaseName*>.
XBM0L	Specified authentication scheme class <*className*> does implement the authentication interface <*interfaceName*>.
XBM0M	Error creating instance of authentication scheme class <*className*>.
XBM0N	JDBC Driver registration with java.sql.DriverManager failed. See next exception for details.
XBM0P	Service provider is read-only. Operation not permitted.
XBM0Q	File <*fileName*> not found. Please make sure that backup copy is the correct one and it is not corrupted.
XBM0R	Unable to remove file <*fileName*>.
XBM0S	Unable to rename file '<*fileName*>' to '*fileName*>'
XBM0T	Ambiguous sub-protocol for database name <*databaseName*>.
XBM0U	No class was registered for identifier <*identifierName*>.
XBM0V	An exception was thrown while loading class <*className*> registered for identifier <*identifierName*>.

SQLSTATE	Message Text
XBM0W	An exception was thrown while creating an instance of class <*className*> registered for identifier <*identifierName*>.
XBM0X	Supplied territory description '<*value*>' is invalid, expecting ln[_CO[_variant]] ln=lower-case two-letter ISO-639 language code, CO=upper-case two-letter ISO-3166 country codes, see java.util.Locale.
XBM0Y	Backup database directory <*directoryName*> not found. Please make sure that the specified backup path is right.
XBM0Z	Unable to copy file '<*value*>' to '<*value*>'. Please make sure that there is enough space and permissions are correct.

Table1. Class Code XCL: Non-SQLSTATE

SQLSTATE	Message Text
XCL01	ResultSet does not return rows. Operation <*operationName*> not permitted.
XCL05	Activation closed. Operation <*operationName*> not permitted.
XCL07	Cursor '<*cursorName*>' is closed. Verify that autocommit is OFF.
XCL08	Cursor '<*cursorName*>' is not on a row.
XCL09	An Activation was passed to the '<*methodName*>' method that does not match the PreparedStatement.
XCL10	A PreparedStatement has been recompiled and the parameters have changed. If you are using JDBC you must prepare the statement again.
XCL12	An attempt was made to put a data value of type '<*type*>' into a data value of type '<*type*>'.
XCL13	The parameter position '<*parameterPosition*>' is out of range. The number of parameters for this prepared statement is '<*number*>'.
XCL14	The column position '<*value*>' is out of range. The number of columns for this ResultSet is '<*number*>'.
XCL15	A ClassCastException occurred when calling the compareTo() method on an object '<*object*>'. The parameter to compareTo() is of class '*className*>'.
XCL16	ResultSet not open. Operation '<*operation*>' not permitted. Verify that autocommit is OFF.
XCL17	Statement not allowed in this database.
XCL19	Missing row in table '<*tableName*>' for key '<*key*>'.
XCL20	Catalogs at version level '<*versionLevel*>' cannot be upgraded to version level '<*versionLevel*>'.
XCL21	You are trying to execute a Data Definition statement (CREATE, DROP, or ALTER) while preparing a different statement. This is not allowed. It can happen if you execute a Data Definition statement from within a static initializer of a Java class that is being used from within a SQL statement.
XCL22	Parameter <*parameterName*> cannot be registered as an OUT parameter because it is an IN parameter.
XCL23	SQL type number '<*type*>' is not a supported type by registerOutParameter().
XCL24	Parameter <*parameterName*> appears to be an output parameter, but it has not been so designated by registerOutParameter(). If it is not an output parameter, then it has to be set to type <*type*>.
XCL25	Parameter <*parameterName*> cannot be registered to be of type <*type*> because it maps to type <*type*> and they are incompatible.

SQLSTATE	Message Text
XCL26	Parameter <*parameterName*> is not an output parameter.
XCL27	Return output parameters cannot be set.
XCL30	An IOException was thrown when reading a '<*value*>' from an InputStream.
XCL31	Statement closed.
XCL33	The table cannot be defined as a dependent of table <*tableName*> because of delete rule restrictions. (The relationship is self-referencing and a self-referencing relationship already exists with the SET NULL delete rule.)
XCL34	The table cannot be defined as a dependent of table <*tableName*> because of delete rule restrictions. (The relationship forms a cycle of two or more tables that cause the table to be delete-connected to itself (all other delete rules in the cycle would be CASCADE)).
XCL35	The table cannot be defined as a dependent of table <*tableName*> because of delete rule restrictions. (The relationship causes the table to be delete-connected to the indicated table through multiple relationships and the delete rule of the existing relationship is SET NULL).
XCL36	The delete rule of foreign key must be <*ruleName*>. (The referential constraint is self-referencing and an existing self-referencing constraint has the indicated delete rule (NO ACTION, RESTRICT or CASCADE).)
XCL37	The delete rule of foreign key must be <*ruleName*>. (The referential constraint is self-referencing and the table is dependent in a relationship with a delete rule of CASCADE.)
XCL38	The delete rule of foreign key must be <*ruleName*>. (The relationship would cause the table to be delete-connected to the same table through multiple relationships and such relationships must have the same delete rule (NO ACTION, RESTRICT or CASCADE).)
XCL39	The delete rule of foreign key cannot be CASCADE. (A self-referencing constraint exists with a delete rule of SET NULL, NO ACTION or RESTRICT.)
XCL40	The delete rule of foreign key cannot be CASCADE. (The relationship would form a cycle that would cause a table to be delete-connected to itself. One of the existing delete rules in the cycle is not CASCADE, so this relationship may be definable if the delete rule is not CASCADE.)
XCL41	the delete rule of foreign key can not be CASCADE. (The relationship would cause another table to be delete-connected to the same table through multiple paths with different delete rules or with delete rule equal to SET NULL.)
XCL42	CASCADE
XCL43	SET NULL
XCL44	RESTRICT
XCL45	NO ACTION
XCL46	SET DEFAULT
XCL47	Use of '<*value*>' requires database to be upgraded from version <*versionNumber*> to version <*versionNumber*> or later.
XCL48	TRUNCATE TABLE is not permitted on '<*value*>' because unique/primary key constraints on this table are referenced by enabled foreign key constraints from other tables.
XCL49	TRUNCATE TABLE is not permitted on '<*value*>' because it has an enabled DELETE trigger (<*value*>).
XCL50	Upgrading the database from a previous version is not supported. The database being accessed is at version level '*versionNumber*>', this software is at version level

SQLSTATE	Message Text
	versionNumber>'.
XCL51	The requested function cannot reference tables in SESSION schema.

Table1. Class Code XCW: Upgrade unsupported

SQLSTATE	Message Text
XCW00	Unsupported upgrade from '<*value*>' to '<*value*>'.

Table1. Class Code XCXA: ID Parse Error

SQLSTATE	Message Text
XCXA0	Invalid identifier.

Table1. Class Code XCXB: DB_Class_Path_Parse_Error

SQLSTATE	Message Text
XCXB0	Invalid database classpath: '<*classpath*>'.

Table1. Class Code XCXC: ID List Parse Error

SQLSTATE	Message Text
XCXC0	Invalid id list.

Table1. Class Code XCXE: No locale

SQLSTATE	Message Text
XCXE0	You are trying to do an operation that uses the territory of the database, but the database does not have a territory.

Table1. Class Code XCY: Properties

SQLSTATE	Message Text
XCY00	Invalid value for property '<*value*>'='<*value*>'.
XCY02	The requested property change is not supported '<*value*>'='<*value*>'.
XCY03	Required property '<*propertyName*>' has not been set.

Table1. Class Code XCZ: org.apache.derby.database.UserUtility

SQLSTATE	Message Text
XCZ00	Unknown permission '<*permissionName*>'.
XCZ01	Unknown user '<*userName*>'.
XCZ02	Invalid parameter '<*value*>'='<*value*>'.

Table1. Class Code XD00x: Dependency Manager

SQLSTATE	Message Text
XD003	Unable to restore dependency from disk. DependableFinder = '<*value*>'. Further

SQLSTATE	Message Text
	information: '<*value*>'.
XD004	Unable to store dependencies.

Table1. Class Code XIE: Import/Export

SQLSTATE	Message Text
XIE01	Connection was null.
XIE03	Data found on line <*lineNumber*> for column <*columnName*> after the stop delimiter.
XIE04	Data file not found: <*fileName*>.
XIE05	Data file cannot be null.
XIE06	Entity name was null.
XIE07	Field and record separators cannot be substrings of each other.
XIE08	There is no column named: <*columnName*>.
XIE09	The total number of columns in the row is: <*number*>.
XIE0B	Column '<*columnName*>' in the table is of type <*type*>, it is not supported by the import/export feature.
XIE0D	Cannot find the record separator on line <*lineNumber*>.
XIE0E	Read endOfFile at unexpected place on line <*lineNumber*.
XIE0I	An IOException occurred while writing data to the file.
XIE0J	A delimiter is not valid or is used more than once.
XIE0K	The period was specified as a character string delimiter.
XIE0M	Table '<*tableName*>' does not exist.

Table1. Class Code XJ: Connectivity Errors

SQLSTATE	Message Text
XJ004	Database '<*databaseName*>' not found.
XJ009	Use of CallableStatement required for stored procedure call or use of output parameters: <*value*>
XJ010	Cannot issue savepoint when autoCommit is on.
XJ011	Cannot pass null for savepoint name.
XJ012	'<*value*>' already closed.
XJ013	No ID for named savepoints.
XJ014	No name for un-named savepoints.
XJ015	Derby system shutdown.
XJ016	Method '<*methodName*>' not allowed on prepared statement.
XJ017	No savepoint command allowed inside the trigger code.
XJ018	Column name cannot be null.
XJ020	Object type not convertible to TYPE '<*typeName*>', invalid java.sql.Types value, or object was null.
XJ022	Unable to set stream: '<*name*>'.
XJ023	Input stream held less data than requested length.
XJ025	Input stream cannot have negative length.

SQLSTATE	Message Text
XJ028	The URL '<*urlValue*>' is not properly formed.
XJ030	Cannot set AUTOCOMMIT ON when in a nested connection.
XJ040	Failed to start database '<*databaseName*>', see the next exception for details.
XJ041	Failed to create database '<*databaseName*>', see the next exception for details.
XJ042	'<*value*>' is not a valid value for property '<*propertyName*>'.
XJ044	'<*value*>' is an invalid scale.
XJ045	Invalid or (currently) unsupported isolation level, '<*value*>', passed to Connection.setTransactionIsolationLevel(). The currently supported values are java.sql.Connection.TRANSACTION_SERIALIZABLE, java.sql.Connection.TRANSACTION_REPEATABLE_READ, java.sql.Connection.TRANSACTION_READ_COMMITTED, and java.sql.Connection.TRANSACTION_READ_UNCOMMITTED.
XJ049	Conflicting create attributes specified.
XJ04B	Batch cannot contain a command that attempts to return a result set.
XJ04C	CallableStatement batch cannot contain output parameters.
XJ056	Cannot set AUTOCOMMIT ON when in an XA connection.
XJ057	Cannot commit a global transaction using the Connection. Commit processing must go through XAResource interface.
XJ058	Cannot rollback a global transaction using the Connection, commit processing must go through XAResource interface.
XJ059	Cannot close a connection while a global transaction is still active.
XJ05B	JDBC attribute '<*attributeName*>' has an invalid value '<*value*>', Valid values are '<*value*>'.
XJ05C	Cannot set holdability ResultSet.HOLD_CURSORS_OVER_COMMIT for a global transaction.
XJ061	The '<*valueName*>' method is only allowed on scroll cursors.
XJ062	Invalid parameter value '<*value*>' for ResultSet.setFetchSize(int rows).
XJ063	Invalid parameter value '<*value*>' for Statement.setMaxRows(int maxRows). Parameter value must be >= 0.
XJ064	Invalid parameter value '<*value*>' for setFetchDirection(int direction).
XJ065	Invalid parameter value '<*value*>' for Statement.setFetchSize(int rows).
XJ066	Invalid parameter value '<*value*>' for Statement.setMaxFieldSize(int max).
XJ067	SQL text pointer is null.
XJ068	Only executeBatch and clearBatch allowed in the middle of a batch.
XJ069	No SetXXX methods allowed in case of USING execute statement.
XJ070	Negative or zero position argument '<*argument*>' passed in a Blob or Clob method.
XJ071	Zero or negative length argument '<*argument*>' passed in a BLOB or CLOB method.
XJ072	Null pattern or searchStr passed in to a BLOB or CLOB position method.
XJ073	The data in this BLOB or CLOB is no longer available. The BLOB or CLOB's transaction may be committed, or its connection is closed.
XJ076	The position argument '<*argument*>' exceeds the size of the BLOB/CLOB.
XJ077	Got an exception when trying to read the first byte/character of the BLOB/CLOB pattern using getBytes/getSubString.
XJ080	USING execute statement passed <*numparameters*> parameters rather than

SQLSTATE	Message Text
	numparameters>.
XJ081	Conflicting create/restore/recovery attributes specified.
XJ081	Invalid value '<*value*>' passed as parameter '<*parameterName*>' to method '<*moduleName*>'.

Table1. Class Code XSAIx: Store - access.protocol.interface statement exceptions

SQLSTATE	Message Text
XSAI2	The conglomerate (<*value*>) requested does not exist.
XSAI3	Feature not implemented.

Table1. Class Code XSAMx: Store - AccessManager

SQLSTATE	Message Text
XSAM0	Exception encountered while trying to boot module for '<*value*>'.
XSAM2	There is no index or conglomerate with conglom id '<*conglomID*>' to drop.
XSAM3	There is no index or conglomerate with conglom id '<*conglomID*>'.
XSAM4	There is no sort called '<*sortName*>'.
XSAM5	Scan must be opened and positioned by calling next() before making other calls.
XSAM6	Record <*recordnumber*> on page <*page*> in container <*containerName*> not found.

Table1. Class Code XSASx: Store-Sort

SQLSTATE	Message Text
XSAS0	A scan controller interface method was called which is not appropriate for a scan on a sort.
XSAS1	An attempt was made to fetch a row before the beginning of a sort or after the end of a sort.
XSAS3	The type of a row inserted into a sort does not match the sorts template.
XSAS6	Could not acquire resources for sort.

Table1. Class Code XSAXx: Store - access.protocol.XA statement exception

SQLSTATE	Message Text
XSAX0	XA protocol violation.
XSAX1	An attempt was made to start a global transaction with an Xid of an existing global transaction.

Table1. Class Code XSCBx: Store-BTree

SQLSTATE	Message Text
XSCB0	Could not create container.
XSCB1	Container <*containerName*> not found.
XSCB2	The required property <*propertyName*> not found in the property list given to createConglomerate() for a btree secondary index.
XSCB3	Unimplemented feature.

SQLSTATE	Message Text
XSCB4	A method on a btree open scan was called prior to positioning the scan on the first row (that is, no next() call has been made yet). The current state of the scan is (<*value*>).
XSCB5	During logical undo of a btree insert or delete, the row could not be found in the tree.
XSCB6	Limitation: Record of a btree secondary index cannot be updated or inserted due to lack of space on the page. Use the parameters derby.storage.pageSize and/or derby.storage.pageReservedSpace to work around this limitation.
XSCB7	An internal error was encountered during a btree scan - current_rh is null = <*value*>, position key is null = <*value*>.
XSCB8	The btree conglomerate <*value*> is closed.
XSCB9	Reserved for testing.

Table1. Class Code XSCG0: Conglomerate

SQLSTATE	Message Text
XSCG0	Could not create a template.

Table1. Class Code XSCHx: Heap

SQLSTATE	Message Text
XSCH0	Could not create container.
XSCH1	Container <*containerName*> not found.
XSCH4	Conglomerate could not be created.
XSCH5	In a base table there was a mismatch between the requested column number <*columnnumber*> and the maximum number of columns <*maxcol*>.
XSCH6	The heap container with container id <*containerID*> is closed.
XSCH7	The scan is not positioned.
XSCH8	The feature is not implemented.

Table1. Class Code XSDAx: RawStore - Data.Generic statement exceptions

SQLSTATE	Message Text
XSDA1	An attempt was made to access an out of range slot on a page.
XSDA2	An attempt was made to update a deleted record.
XSDA3	Limitation: Record cannot be updated or inserted due to lack of space on the page. Use the parameters derby.storage.pageSize and/or derby.storage.pageReservedSpace to work around this limitation.
XSDA4	An unexpected exception was thrown.
XSDA5	An attempt was made to undelete a record that was not deleted.
XSDA6	Column <*column*> of row is null, it needs to be set to point to an object.
XSDA7	Restore of a serializable or SQLData object of class <*className*>, attempted to read more data than was originally stored.
XSDA8	Exception during restore of a serializable or SQLData object of class <*className*>.
XSDA9	Class not found during restore of a serializable or SQLData object of class <*className*>.
XSDAA	Illegal time stamp <*timestamp*>, either time stamp is from a different page or of incompatible implementation.

SQLSTATE	Message Text
XSDAB	Cannot set a null time stamp.
XSDAC	Attempt to move either rows or pages from one container to another.
XSDAD	Attempt to move zero rows from one page to another.
XSDAE	Can only make a record handle for special record handle IDs.
XSDAF	Using special record handle as if it were a normal record handle.
XSDAG	The allocation nested top transaction cannot open the container
XSDAI	Page <*page*> being removed is already locked for deallocation.
XSDAJ	Exception during write of a serializable or SQLData object.
XSDAK	The wrong page was retrieved for record handle <*value*>.
XSDAL	Record handle <*value*> unexpectedly points to overflow page.

Table1. Class Code XSDBx: RawStore - Data.Generic transaction exceptions

SQLSTATE	Message Text
XSDB0	Unexpected exception on in-memory page <*page*>.
XSDB1	Unknown page format at page <*page*>.
XSDB2	Unknown container format at container <*containerName*> : <*value*>.
XSDB3	Container information cannot change once written: was <*value*>, now <*value*>.
XSDB4	Page <*page*> is at version <*value*> but the log file contains change version <*value*>. Either the log records for this page are missing or this page was not written to disk properly.
XSDB5	Log has change record on page <*page*>, which is beyond the end of the container.
XSDB6	Another instance of Derby may have already booted the database <*value*>.
XSDB7	WARNING: Derby (instance <*value*>) is attempting to boot the database <*value*> even though Derby (instance <*value*>) might still be active. Only one instance of Derby should boot a database at a time. Severe and non-recoverable corruption can result and might have already occurred.
XSDB8	WARNING: Derby (instance <*value*>) is attempting to boot the database <*value*> even though Derby (instance <*value*>) might still be active. Only one instance of Derby should boot a database at a time. Severe and non-recoverable corruption can result if two instances of Derby boot on the same database at the same time. The db2j.database.forceDatabaseLock=true property is set so the database will not boot until the db.lck is no longer present. Normally this file is removed when the first instance of Derby to boot on the database exits. However, it is not removed in some shutdowns. If the file is not removed, you must remove it manually. It is important to verify that no other VM is accessing the database before manually deleting the db.lck file.
XSDB9	Stream container <*containerName*> is corrupt.
XSDBA	Attempt to allocate object <*object*> failed.

Table1. Class Code XSDFx: RawStore - Data.Filesystem statement exceptions

SQLSTATE	Message Text
XSDF0	Could not create file <*fileName*> as it already exists.
XSDF1	Exception during creation of file <*fileName*> for container.
XSDF2	Exception during creation of file <*fileName*> for container, file could not be removed.

SQLSTATE	Message Text
	The exception was: <*value*>.
XSDF3	Cannot create segment <*segmentName*>.
XSDF4	Exception during remove of file <*fileName*> for dropped container, file could not be removed <*value*>.
XSDF6	Cannot find the allocation page <*page*>.
XSDF7	Newly created page failed to be latched <*value*>.
XSDF8	Cannot find page <*page*> to reuse.
XSDFB	Operation not supported by a read only database.
XSDFD	Different page image read on two I/Os on Page <*page*>. The first image has an incorrect checksum, the second image has a correct checksum. Page images follows: <*value*> <*value*>.
XSDFF	The requested operation failed due to an unexpected exception.

Table1. Class Code XSDGx: RawStore - Data.Filesystem database exceptions

SQLSTATE	Message Text
XSDG0	Page <*page*> could not be read from disk.
XSDG1	Page <*page*> could not be written to disk, please check if disk is full.
XSDG2	Invalid checksum on Page <*page*>, expected=<*value*>, on-disk version=<*value*>, page dump follows: <*value*>.
XSDG3	Meta-data for Container <*containerName*> could not be accessed.
XSDG5	Database is not in create mode when createFinished is called.
XSDG6	Data segment directory not found in <*value*> backup during restore. Please make sure that backup copy is the right one and it is not corrupted.
XSDG7	Directory <*directoryName*> could not be removed during restore. Please make sure that permissions are correct.
XSDG8	Unable to copy directory '<*directoryName*>' to '<*value*>' during restore. Please make sure that there is enough space and permissions are correct.

Table1. Class Code XSLAx: RawStore - Log.Generic database exceptions

SQLSTATE	Message Text
XSLA0	Cannot flush the log file to disk <*value*>.
XSLA1	Log Record has been sent to the stream, but it cannot be applied to the store (Object <*object*>). This may cause recovery problems also.
XSLA2	An IOException occurred while accessing the log file. The system will shut down.
XSLA3	The log file is corrupt. The log stream contains invalid data.
XSLA4	Unable to write to the log, most likely because the log is full. It is also possible that the file system is read-only, the disk failed, or another problem occurred with the media. Delete unnecessary files.
XSLA5	Cannot read log stream for some reason to rollback transaction <*value*>.
XSLA6	Cannot recover the database.
XSLA7	Cannot redo operation <*operation*> in the log.
XSLA8	Cannot rollback transaction <*value*>, trying to compensate <*value*> operation with <*value*>.

SQLSTATE	Message Text
XSLAA	The store has been marked for shutdown by an earlier exception.
XSLAB	Cannot find log file <*logfileName*>. Verify that the logDevice property is set with the correct path separator for your platform.
XSLAC	Database at <*value*> have incompatible format with the current version of software, it may have been created by or upgraded by a later version.
XSLAD	Log record at instance <*value*> in log file <*logfileName*> is corrupted. Expected log record length <*value*>, actual length <*value*>.
XSLAE	Control file at <*value*> cannot be written or updated.
XSLAF	A read-only database was created with dirty data buffers.
XSLAH	A read-only database is being updated.
XSLAI	Cannot log the checkpoint log record.
XSLAJ	The log record size <*value*> exceeds the maximum allowable log file size <*maxSize*>. An error was encountered in log file <*fileName*>, position <*value*>.
XSLAK	Database has exceeded largest log file number <*value*>.
XSLAL	The log record size <*value*> exceeds the maximum allowable log file size <*maxSize*>. An error was encountered in log file <*fileName*>, position <*value*>.
XSLAM	Cannot verify database format at <*value*> due to IOException.
XSLAN	Database at <*value*> has an incompatible format with the current version of the software. The database was created by or upgraded by version <*version*>.
XSLAO	Recovery failed. Unexpected problem <*value*>.
XSLAP	Database at <*value*> is at version <*version*>. Beta databases cannot be upgraded.
XSLAQ	Cannot create log file at directory <*directory*>.
XSLAR	Unable to copy log file '<*logfileName*>' to '<*value*>' during restore. Please make sure that there is enough space and permissions are correct.
XSLAS	Log directory <*directory*> not found in backup during restore. Please make sure that backup copy is the correct one and it is not corrupted.
XSLAT	Log directory <*directory*> exists. Please make sure specified logDevice location is correct.

Table1. Class Code XSLBx: RawStore - Log.Generic statement exceptions

SQLSTATE	Message Text
XSLB1	Log operation <*logoperation*> encounters error writing itself out to the log stream, this could be caused by an errant log operation or internal log buffer full due to excessively large log operation.
XSLB2	Log operation <*logoperation*> logging excessive data, it filled up the internal log buffer.
XSLB4	Cannot find truncationLWM <*value*>.
XSLB5	Illegal truncationLWM instance <*value*> for truncation point <*value*>. Legal range is from <*value*> to <*value*>.
XSLB6	Trying to log a 0 or -ve length log Record.
XSLB8	Trying to reset a scan to <*value*>, beyond its limit of <*value*>.
XSLB9	Unable to issue any more changes. Log factory has been stopped.

Table1. Class Code XSRSx: RawStore - protocol.Interface statement exceptions

SQLSTATE	Message Text
XSRS0	Cannot freeze the database after it is already frozen.
XSRS1	Cannot backup the database to <*value*>, which is not a directory.
XSRS4	Error renaming file (during backup) from <*value*> to <*value*>.
XSRS5	Error copying file (during backup) from <*path*> to <*path*>.
XSRS6	Cannot create backup directory <*directoryName*>.
XSRS7	Backup caught unexpected exception.
XSRS8	Log device can only be set during database creation time, it cannot be changed after the database is created.
XSRS9	Record <*recordName*> no longer exists.

Table1. Class Code XSTA2: XACT_TRANSACTION_ACTIVE

SQLSTATE	Message Text
XSTA2	A transaction was already active when an attempt was made to activate another transaction.

Table1. Class Code XSTBx: RawStore - Transactions.Basic system exceptions

SQLSTATE	Message Text
XSTB0	An exception was thrown during transaction abort.
XSTB2	Unable to log transaction changes, possibly because the database is read-only.
XSTB3	Cannot abort transaction because the log manager is null, probably due to an earlier error.
XSTB5	Creating database with logging disabled encountered unexpected problem.
XSTB6	Cannot substitute a transaction table with another while one is already in use.

Table1. Class Code XXXXX : No SQLSTATE

SQLSTATE	Message Text
XXXXX	Normal database session close.

JDBC Reference

Derby comes with a built-in JDBC driver. That makes the JDBC API the only API for working with Derby databases. The driver is a native protocol all-Java driver (type number four of types defined by Sun).

This chapter provides reference information about Derby's implementation of the JDBC API and documents the way it conforms to the JDBC 2.0 and 3.0 APIs.

See the *Derby Developer's Guide* for task-oriented instructions on working with the driver.

This JDBC driver implements the standard JDBC interface defined by Sun. When invoked from an application running in the same JVM as Derby, the JDBC driver supports connections to a Derby database in embedded 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 Derby, sends requests to Derby through the embedded JDBC driver.

The Derby JDBC implementation provides access to Derby 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 Derby.

Core JDBC java.sql Classes, Interfaces, and Methods

This section details Derby's implementation of the following *java.sql* classes, interfaces, and methods:

- *java.sql.Driver*
- *java.sql.DriverManager.getConnection*
- *java.sql.Driver.getPropertyInfo*
- *java.sql.Connection*
- *java.sql.DatabaseMetaData*
- *java.sql.Statement*
- *java.sql.PreparedStatement*
- *java.sql.CallableStatement*
- *java.sql.ResultSet*
- *java.sql.ResultSetMetaData*
- *java.sql.SQLException*
- *java.sql.SQLWarning*
- *Mapping of java.sql.Types to SQL types*

java.sql.Driver

The class that loads Derby's local JDBC driver is the class *org.apache.derby.jdbc.EmbeddedDriver*. Listed below are some of the ways to create instances of that class. Do not use the class directly through the *java.sql.Driver* interface. Use the *DriverManager* class to create connections.

- *Class.forName("org.apache.derby.jdbc.EmbeddedDriver")*

  Our recommended manner, because it ensures that the class is loaded in all JVMs by creating an instance at the same time.

- *new org.apache.derby.jdbc.EmbeddedDriver()*

  Same as *Class.forName("org.apache.derby.jdbc.EmbeddedDriver")* , except that it requires the class to be found when the code is compiled.

- *Class c = org.apache.derby.jdbc.EmbeddedDriver.class*

This is also the same as *Class.forName("org.apache.derby.jdbc.EmbeddedDriver")* , 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.

```
java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
    applicationClass
```

The actual driver that gets registered in the *DriverManager* to handle the *jdbc:derby:* protocol is not the class *org.apache.derby.jdbc.EmbeddedDriver*; that class simply detects the type of Derby driver needed and then causes the appropriate Derby driver to be loaded.

The only supported way to connect to a Derby system through the *jdbc:derby:* protocol is using the *DriverManager* to obtain a driver (*java.sql.Driver*) or connection (*java.sql.Connection*) through the *getDriver* and *getConnection* method calls.

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, Derby checks whether they are valid for the current system if user authentication is enabled. User names are passed to Derby 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, Derby sets the default user to *APP*, which Derby uses to name the default schema. If a user is supplied, the default schema is the same as the user name.

Derby database connection URL syntax

A Derby 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" in Chapter 1 of the *Derby Developer's Guide* .

Syntax of database connection URLs for applications with embedded databases

For applications with embedded databases, the syntax of the database connection URL is

```
jdbc:derby: [
subsubprotocol:
][databasename][;
attributes
]*
```

- *jdbc:derby:*

  In JDBC lingo, *derby* is the *subprotocol* for connecting to a Derby database. The subprotocol is always *derby* and does not vary.

- *subsubprotocol:*

  subsubprotocol, which is not typically specified, specifies where Derby looks for a database: in a directory, in a classpath, 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 classpath 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 *Derby 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 specify the database name alone, or with a relative or absolute path. See "Standard Connections-Connecting to Databases in the File System" in Chapter 1 of the *Derby Developer's Guide* .

- *attributes*

  Specify 0 or more database connection URL attributes as detailed in Attributes of the Derby database connection URL .

Additional SQL syntax

Derby also supports the following SQL standard syntax to obtain a reference to the current connection in a database-side JDBC procedure or method:

```
jdbc:default:connection
```

Attributes of the Derby database connection URL

You can supply an optional list of attributes to a database connection URL. Derby

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 Derby and are listed in Setting attributes for the database connection URL .

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.

```
import java.util.Properties;

Connection conn = DriverManager.getConnection(
    "jdbc:derby:sampleDB;create=true");
-- setting an attribute in a Properties object
Properties myProps = new Properties();
myProps.put("create", "true");
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sampleDB", myProps);
-- passing user name and password
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sampleDB", "dba", "password");
```

Note: Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored. (Derby does provide a tool for parsing the correctness of attributes. For more information, see the *Derby Tools and Utilities Guide* .)

java.sql.Driver.getPropertyInfo

To get the *DriverPropertyInfo* object, request the JDBC driver from the driver manager:

```
java.sql.DriverManager.getDriver("jdbc:derby:").
    getPropertyInfo(
URL

prop
)
```

Do not request it from *org.apache.derby.jdbc.EmbeddedDriver*, which is only an intermediary class that loads the actual driver.

This method might return a *DriverPropertyInfo* object. In a Derby system, it consists of an array of database connection URL attributes. The most useful attribute is *databaseName=nameofDatabase* , which means that the object consists of a list of booted databases in the current system.

For example, if a Derby system has the databases *toursDB* and *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 `A:/dbs/tours94`, the array returned from *getPropertyInfo* contains one object corresponding to the *databaseName* attribute. The choices field of the *DriverPropertyInfo* object will contain an array of three Strings with the values *toursDB*, *flightsDB*, and `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 *java.sql.Driver.getPropertyInfo*, see "Offering Connection Choices to the User" in Chapter 8 of the *Derby Developer's Guide* .

java.sql.Connection

A Derby *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 Derby system to shut down, which not only closes the connection but means that no new connections should be created in the current JVM.

java.sql.Connection.setTransactionIsolation

java.sql.Connection.TRANSACTION_SERIALIZABLE, *java.sql.Connection.TRANSACTION_REPEATABLE_READ*, *java.sql.Connection.TRANSACTION_READ_COMMITTED, and java.sql.Connection.TRANSACTION_READ_UNCOMMITTED* transaction isolations are available from a Derby database.

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 Derby properties), and *Connections* to databases on read-only media return true.

Connection functionality not supported

Derby 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

This section discuss java.sql.DatabaseMetaData functionality in Derby.

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)*.

getProcedureColumns

Derby supports Java procedures. Derby allows you to call Java procedures within SQL statements. Derby returns information about the parameters in the *getProcedureColumns* call. If the corresponding Java method is overloaded, it returns information about each signature separately. Derby returns information for all Java procedures defined by CREATE PROCEDURE.

getProcedureColumns returns a *ResultSet*. Each row describes a single parameter or return value.

Parameters to getProcedureColumns

The JDBC API defines the following parameters for this method call:

- *catalog*

  always use *null* for this parameter in Derby.
- *schemaPattern*

  Java procedures have a schema.
- *procedureNamePattern*

  a String object representing a procedure name pattern.
- *column-Name-Pattern*

  a String object representing the name pattern of the parameter names or return value names. Java procedures have parameter names matching those defined in the CREATE PROCEDURE statement. Use "%" to find all parameter names.

Columns in the ResultSet returned by *getProcedureColumns*

Columns in the *ResultSet* returned by *getProcedureColumns* are as described by the API. Further details for some specific columns:

- PROCEDURE_CAT

  always "null" in Derby
- PROCEDURE_SCHEM

  schema for a Java procedure
- PROCEDURE_NAME

  the name of the procedure
- COLUMN_NAME

  the name of the parameter (see column-Name-Pattern)
- COLUMN_TYPE

short indicating what the row describes. Always is *DatabaseMetaData.procedureColumnIn* for method parameters, unless the parameter is an array. If so, it is *DatabaseMetaData.procedureColumnInOut*. It always returns *DatabaseMetaData.procedureColumnReturn* for return values.

- TYPE_NAME

  Derby-specific name for the type.

- NULLABLE

  always returns *DatabaseMetaData.procedureNoNulls* for primitive parameters and *DatabaseMetaData.procedureNullable* for object parameters

- REMARKS

  a String describing the java type of the method parameter

- METHOD_ID

  a Derby-specific column.

- PARAMETER_ID

  a Derby-specific column.

DatabaseMetaData functionality not supported

In the current release, Derby 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*
- *getTablePrivileges*

Derby 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 might not necessarily identify a unique row in the table.

java.sql.Statement

Derby does not implement the following JDBC 1.2 methods of *java.sql.Statement*:

- *cancel*
- *setEscapeProcessing*
- *setQueryTimeout*

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, INTERSECT, EXCEPT, or UNION.

Closing a *Statement* causes all open *ResultSet* objects 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.PreparedStatement

Derby 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.

Prepared statements and 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 amounts 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.

Derby 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. 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.

According to the JDBC standard, streams can be stored only in columns of the data types shown in Streamable JDBC Data Types . Streams cannot be stored in columns of the other built-in data types or of user-defined data types.

Table1. Streamable JDBC Data Types

Column Values	Type Correspondent	AsciiStream	UnicodeStream	BinaryStream
CLOB	java.sql.Clob	x	x	'
CHAR	'	x	x	'
VARCHAR	'	x	x	'
LONGVARCHAR	'	X	X	'
BINARY	'	x	x	x
BLOB	java.sql.Blob	x	x	x
VARBINARY	'	x	x	x
LONGVARBINARY	'	x	x	X

A large X indicates the preferred target data type for the type of stream. (See Mapping of java.sql.Types to SQL Types .)

Note: If the stream is stored in a column of a type other than LONG VARCHAR or LONG VARCHAR FOR BIT DATA, the entire stream must be able to fit into memory at one time. Streams stored in LONG VARCHAR and LONG VARCHAR FOR BIT DATA 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:

```
Statement s = conn.createStatement();
s.executeUpdate("CREATE TABLE atable (a INT, b LONG VARCHAR)");
conn.commit();
java.io.File file = new java.io.File("derby.txt");
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

Derby supports all the JDBC 1.2 methods of *CallableStatement*:

- *getBoolean()*
- *getByte()*
- *getBytes()*
- *getDate()*
- *getDouble()*
- *getFloat()*
- *getInt()*
- *getLong()*
- *getObject()*
- *getShort()*
- *getString()*
- *getTime()*
- *getTimestamp()*

- *registerOutParamter()*
- *wasNull()*

CallableStatements and OUT Parameters

Derby supports OUT parameters and CALL statements that return values, as in the following example:

```
CallableStatement cs = conn.prepareCall(
   "? = CALL getDriverType(cast (? as INT))"
cs.registerOutParameter(1, Types.INTEGER);
cs.setInt(2, 35);
cs.executeUpdate();
```

Note: Using a CALL statement with a procedure 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 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 SQL standard.

Given the following example:

```
CallableStatement call = conn.prepareCall(
    "{CALL doubleMyInt(?)}");
// for inout parameters, it is good practice to
// register the outparameter before setting the input value
call.registerOutParameter(1, Types.INTEGER);
call.setInt(1,10);
call.execute();
int retval = call.getInt(1);
```

The method *doubleIt* should take a one-dimensional array of `ints`. Here is sample source code for that method:

```
public static void doubleMyInt(int[] i) {
    i[0] *=2;
    /* Derby  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.

Table1. INOUT Parameter Type Correspondence

JDBC Type	Array Type for Method Parameter	Value and Return Type
BIGINT	long[]	long
BINARY	byte[][]	byte[]
BIT	boolean[]	boolean
DATE	*java.sql.Date[]*	*java.sql.Date*
DOUBLE	double[]	double
FLOAT	double[]	double
INTEGER	int[]	int

JDBC Type	Array Type for Method Parameter	Value and Return Type
LONGVARBINARY	byte[][]	byte[]
REAL	float[]	float
SMALLINT	short[]	short
TIME	*java.sql.Time[]*	*java.sql.Time*
TIMESTAMP	*java.sql.Timestamp[]*	*java.sql.Timestamp*
VARBINARY	byte[][]	byte[]
OTHER	*yourType[]*	*yourType*
JAVA_OBJECT (only valid in Java2/JDBC 2.0 environments)	*yourType[]*	*yourType*

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 might 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*.

Derby 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*. Derby uses *java.math.BigDecimal.ROUND_HALF_DOWN*.

ResultSets and 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 Streamable JDBC Data Types for a list of types that support the various streams. (See also Mapping of java.sql.Types to SQL Types .)

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:

```
// retrieve data as a stream
ResultSet rs = s.executeQuery("SELECT b FROM atable");
```

```
while (rs.next()) {
    // use a java.io.InputStream to get the data
    java.io.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;
        columnSize += size;
        String chunk = new String(buff, 0, size);
        System.out.print(chunk);
    }
}
rs.close();
s.close();
conn.commit();
```

java.sql.ResultSetMetaData

Derby does not track the source or updatability of columns in *ResultSets*, and so always returns the following constants for the following methods:

Method Name	Value
isDefinitelyWritable	false
isReadOnly	false
isWritable	false

java.sql.SQLException

Derby supplies values for the *getMessage()*, *getSQLState()*, and *getErrorCode()* calls of *SQLExceptions*. In addition, Derby 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 Derby. For information on processing *SQLExceptions*, see "Working with Derby SQLExceptions in an Application" in Chapter 5 of the *Derby Developer's Guide* .

java.sql.SQLWarning

Derby can generate a warning in certain circumstances. A warning is generated if, for example, you try to connect to a database with the *create* attribute set to *true* if the database already exists. Aggregates like sum() also raise a warning if NULL values are encountered during the evaluation.

All other informational messages are written to the Derby system's *derby.log* file.

Mapping of java.sql.Types to SQL types

Mapping of java.sql.Types to SQL Types shows the mapping of *java.sql.Types* to SQL types.

Table1. Mapping of java.sql.Types to SQL Types

java.sql.Types	SQL Types
BIGINT	BIGINT

java.sql.Types	SQL Types
BINARY	CHAR FOR BIT DATA
BIT [1]	CHAR FOR BIT DATA
BLOB	BLOB (JDBC 2.0 and up)
CHAR	CHAR
CLOB	CLOB (JDBC 2.0 and up)
DATE	DATE
DECIMAL	DECIMAL
DOUBLE	DOUBLE PRECISION
FLOAT	DOUBLE PRECISION [2]
INTEGER	INTEGER
LONGVARBINARY	LONG VARCHAR FOR BIT DATA
LONGVARCHAR	LONG VARCHAR
NULL	Not a data type; always a value of a particular type
NUMERIC	DECIMAL
REAL	REAL
SMALLINT	SMALLINT
TIME	TIME
TIMESTAMP	TIMESTAMP
VARBINARY	VARCHAR FOR BIT DATA
VARCHAR	VARCHAR

Notes:

1. BIT is only valid in JDBC 2.0 and earlier environments.
2. 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.

java.sql.Blob and java.sql.Clob

In JDBC 2.0, *java.sql.Blob* is the mapping for the SQL BLOB (binary large object) type; *java.sql.Clob* is the mapping for the SQL CLOB (character large object) type.

java.sql.Blob and java.sql.Clob provide a logical pointer to the large object rather than a complete copy of the objects. Derby processes only one data page into memory at a time. The whole BLOB does not need to be processed and stored in memory just to access the first few bytes of the LOB object

Derby now supports the built-in BLOB or CLOB data types. Derby also provides the following support for these data types:

- **BLOB Features** Derby supports the *java.sql.Blob* interface and the BLOB-related methods in *java.sql.PreparedStatement* and *java.sql.ResultSet* . The *getBlob* methods of *CallableStatement* are not implemented.
- **CLOB Features** Derby supports the java.sql.Clob interface and the CLOB-related methods in *java.sql.PreparedStatement* and *java.sql.ResultSet* . The *getClob* methods of *CallableStatement* procedures are not implemented.

To use the java.sql.Blob and java.sql.Clob features:

- Use the SQL BLOB type for storage; LONG VARCHAR FOR BIT DATA, BINARY, and VARCHAR FOR BIT DATA types also work.
- Use the SQL CLOB type for storage; LONG VARCHAR, CHAR, and VARCHAR types also work.
- Use the *getBlob* or *getClob* methods on the *java.sql.ResultSet* interface to retrieve a *BLOB* or *CLOB* handle to the underlying data.
- You cannot call static methods (SQL extension over SQL) on any LOB-columns.

In addition, casting between strings and BLOBs is not recommended because casting is platform and database dependent.

Derby uses unicode strings (2 byte characters), while other database products may use ASCII characters (1 byte per character). If various codepages are used, each character might need several bytes. A larger BLOB type might be necessary to accommodate a normal string in Derby. You should use CLOB types for storing strings.

Restrictions on BLOB, CLOB, (LOB-types):

- LOB-types cannot be compared for equality(=) and non-equality(!=, <>.
- LOB-typed values are not order-able, so <, <=, >, >= tests are not supported.
- LOB-types cannot be used in indices or as primary key columns.
- DISTINCT, GROUP BY, ORDER BY clauses are also prohibited on LOB-types.
- LOB-types cannot be involved in implicit casting as other base-types.

Derby implements all of the methods for these JDBC 2.0 interfaces except for the set and get methods in *CallableStatement* interface.

Recommendations: Because the lifespan of a *java.sql.Blob* or *java.sql.Clob* ends when the transaction commits, turn off auto-commit with the *java.sql.Blob* or *java.sql.Clob* features.

Table1. JDBC 2.0 java.sql.Blob Methods Supported

Returns	Signature	Implementation Notes
InputStream	*getBinaryStream()*	'
byte[]	*getBytes(long pos, int length)*	Exceptions are raised if *pos* < 1, if *pos* is larger than the length of the , or if *length* <= 0.
long	*length()*	'
long	*position(byte[] pattern, long start)*	Exceptions are raised if *pattern* == null, if *start* < 1, or if *pattern* is an array of length 0.
long	*position(Blob pattern, long start)*	Exceptions are raised if *pattern* == null, if *start* < 1, if *pattern* has length 0, or if an exception is thrown when trying to read the first byte of *pattern*.

Table1. JDBC 2.0 java.sql.Clob Methods Supported

Returns	Signature	Implementation Notes
InputStream	*getAsciiStream()*	'
Reader	*getCharacterStream()*	NOT SUPPORTED
String	*getSubString(long pos, int length)*	Exceptions are raised if *pos* < 1, if *pos* is larger than the length of the *Clob*, or if length <= 0.
long	*length()*	'
long	*position(Clob searchstr, long start)*	Exceptions are raised if *searchStr* == null or *start* <

Returns	Signature	Implementation Notes
		1, if *searchStr* has length 0, or if an exception is thrown when trying to read the first char of *searchStr.*
long	*position(String searchstr, long start)*	Exceptions are raised if *searchStr* == null or start < 1, or if the pattern is an empty string.

Notes

The usual Derby locking mechanisms (shared locks) prevent other transactions from updating or deleting the database item to which the *java.sql.Blob* or *java.sql.Clob* object is a pointer. However, in some cases, Derby's instantaneous lock mechanisms could allow a period of time in which the column underlying the *java.sql.Blob* or *java.sql.Clob* is unprotected. A subsequent call to *getBlob/getClob*, or to a *java.sql.Blob/java.sql.Clob*method, could cause undefined behavior.

Furthermore, there is nothing to prevent the transaction that holds the *java.sql.Blob/java.sql.Clob* (as opposed to another transaction) from updating the underlying row. (The same problem exists with the *getXXXStream* methods.) Program applications to prevent updates to the underlying object while a *java.sql.Blob/java.sql.Clob* is open on it; failing to do this could result in undefined behavior.

Do not call more than one of the *ResultSet getXXX* methods on the same column if one of the methods is one of the following:

- *getBlob*
- *getClob*
- *getAsciiStream*
- *getBinaryStream*
- *getUnicodeStream*

These methods share the same underlying stream; calling one more than one of these methods on the same column so could result in undefined behavior. For example:

```
ResultSet rs = s.executeQuery("SELECT text FROM CLOBS WHERE i = 1");
while (rs.next()) {
    aclob=rs.getClob(1);
    ip = rs.getAsciiStream(1);
}
```

The streams that handle long-columns are not thread safe. This means that if a user chooses to open multiple threads and access the stream from each thread, the resulting behavior is undefined.

Clobs are not locale-sensitive.

java.sql.Connection

Table1. JDBC 2.0 Connection Methods Supported

Returns	Signature
Statement	*createStatement(int resultSetType, int resultSetConcurrency)*
PreparedStatement	*prepareStatement(String sql, int resultSetType, int resultSetConcurrency)*
CallableStatement	*prepareCall(String sql, int resultSetType, int*

Returns	Signature
	resultSetConcurrency

Implementation notes
ResultSet.TYPE_FORWARD_ONLY and *ResultSet.TYPE_SCROLL_INSENSITIVE* are the only scrolling types supported. If you request *TYPE_SCROLL_SENSITIVE*, Derby issues an *SQLWarning* and returns a *TYPE_SCROLL_INSENSITIVE ResultSet*.

These methods support both *ResultSet.CONCUR_READ_ONLY* and *ResultSet.CONCUR_UPDATABLE* concurrencies. However, you can only request an updatable ResultSet that has a *TYPE_FORWARD_ONLY* scrolling type. If you request an updatable ResultSet with *SCROLL_SENSITIVE* or *SCROLL_INSENSITIVE* types, Derby issues an *SQLWarning* and returns *TYPE_SCROLL_INSENSITIVE READ_ONLY ResultSet*.

(Use *Connection.getWarnings* to see warnings.)

java.sql.ResultSet

Table1. JDBC 2.0 ResultSet Methods Supported

Returns	Signature	Implementation Notes
void	*afterLast()*	'
void	*beforeFirst()*	'
void	*beforeFirst()*	'
void	*deleteRow()*	After the row is updated, the ResultSet object will be positioned before the next row. Before issuing any methods other than close on the ResultSet object, the program will need to reposition the ResultSet object by using the *next()* method.
boolean	*first()*	'
Blob	*getBlob(int columnIndex)*	See java.sql.Blob and java.sql.Clob
Blob	*getBlob(String column-Name)*	
Clob	*getClob(int columnIndex)*	
Clob	*getClob(String column-Name)*	
int	*getConcurrency()*	If the Statement object has *CONCUR_READ_ONLY* concurrency, then this method will return *ResultSet.CONCUR_READ_ONLY*. But if the Statement object has *CONCUR_UPDATABLE* concurrency, then the return value will depend on whether the underlying language ResultSet is updatable or not. If the language ResultSet is updatable, then *getConcurrency()* will return *ResultSet.CONCUR_UPDATABLE*. If the language ResultSet is not updatable, then *getConcurrency()* will return *ResultSet.CONCUR_READ_ONLY*.
int	*getFetchDirection()*	'
int	*getFetchSize()*	Always returns 1.
int	*getRow()*	'
boolean	*isAfterLast()*	'

Returns	Signature	Implementation Notes
boolean	*isBeforeFirst*	'
boolean	*isFirst()*	'
boolean	*isLast()*	'
boolean	*last()*	'
boolean	*previous()*	'
boolean	*relative(int rows)*	'
void	*setFetchDirection(int direction)*	'
void	*setFetchSize(int rows)*	A fetch size of 1 is the only size supported.
void	*updateRow()*	After the row is updated, the ResultSet object will be positioned before the next row. Before issuing any methods other than close on the ResultSet object, the program will need to reposition the ResultSet object by using the *next()* method.

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, this method automatically closes the *ResultSet* if it is called and there are no more rows. *afterLast()* does not close the *ResultSet*, for example.

JDBC is not required to have auto-commit off when using updatable ResultSets.

At this moment, Derby does not support the insertRow() method for updatable *ResultSets*.

java.sql.Statement

Table1. JDBC2.0 java.sql.Statement Methods Supported

Returns	Signature	Implementation Notes
void	*addBatch(String sql)*	'
void	*clearBatch()*	'
int[]	*executeBatch()*	'
int	*getFetchDirection()*	Method call does not throw an exception, but call is ignored.
int	*getFetchSize()*	Method call does not throw an exception, but call is ignored.
int	*getMaxFieldSize()*	'
void	*getMaxRows()*	'
void	*setEscapeProcessing(boolean enable)*	'
void	*setFetchDirection(int direction)*	Method call does not throw an exception, but call is ignored.
void	*setFetchSize(int rows)*	Method call does not throw an exception, but call is ignored.
void	*setMaxFieldSize(int max)*	Has no effect on *Blobs* and *Clobs*.
void	*setMaxRows()*	'

java.sql.PreparedStatement

Table1. JDBC 2.0 java.sql.PreparedStatement Methods Supported

Returns	Signature	Implementation Notes
void	*addBatch()*	'
ResultSetMetaData	*getMetaData()*	'
void	*setBlob(int i, Blob x)*	'
void	*setClob(int i, Clob x)*	'

java.sql.CallableStatement

Table1. JDBC 2.0 java.sql.CallableStatements Methods Supported

Returns	Signature	Implementation Notes
BigDecimal	*getBigDecimal*	'
Date	*getDate(int, Calendar)*	'
Time	*getTime(int,Calendar)*	'
Timestamp	*getTimestamp(int, Calendar)*	'

java.sql.DatabaseMetaData

Derby implements all of the JDBC 2.0 methods for this interface.

java.sql.ResultSetMetaData

Derby implements all of the JDBC 2.0 methods for this interface.

java.sql.BatchUpdateException

Thrown if there is a problem with a batch update.

JDBC Package for Connected Device Configuration/Foundation Profile (JSR169)

Derby supports the JDBC API defined for the Connected Device Configuration/Foundation Profile, also known as JSR169. The features supported are a subset of the JDBC 3.0 specification. Support for JSR169 is limited to the embedded driver. Derby does not support using the Network Server under JSR169.

To obtain a connection under JSR169 specifications, use the `org.apache.derby.jdbc.EmbeddedSimpleDataSource` class. This class is identical in implementation to the `org.apache.derby.jdbc.EmbeddedDataSource` class. See the *Derby Developer's Guide* for information on using the properties of the `org.apache.derby.jdbc.EmbeddedDataSource` class.

Some other features to note concerning the JSR169 implementation using Derby:

- Applications must get and set `DECIMAL` values using alternate JDBC `getXXX` and

`setXXX` methods, such as `getString()` and `setString()`. Any alternate method that works against a DECIMAL type with JDBC 2.0 or 3.0 will work in JSR169.
- Java functions and procedures that use server-side JDBC parameters such as `CONTAINS SQL`, `READS SQL DATA` or `MODIFIES SQL DATA` are not supported in JSR169.
- The standard API used to obtain a connection (`jdbc:default:connection`) is not supported in JSR169. A runtime error may occur when the routine tries to obtain a connection using `jdbc:default:connection`.
- Diagnostic tables are not supported.
- Triggers are not supported.
- Encrypted databases are not supported.
- DriverManager is not supported. You cannot use DriverManager.getConnection() to obtain a connection.

JDBC 3.0-only features

JDBC 3.0 adds some functionality to the core API. This section documents the features supported by Derby.
Note: These features are present only in a Java 2 version 1.4 or higher environment.

These features are:
- New DatabaseMetaData methods. See *java.sql.DatabaseMetaData* .
- Retrieval of parameter metadata. See *java.sql.ParameterMetaData and java.sql.PreparedStatement* .
- Retrieval of auto-generated keys. See *java.sql.Statement and java.sql.DatabaseMetaData* .
- Savepoints. See *java.sql.Connection* .
- HOLD Cursors. See *java.sql.DatabaseMetaData* .

The complete list:
- *java.sql.Connection*
- *java.sql.DatabaseMetaData*
- *java.sql.ParameterMetaData*
- *java.sql.PreparedStatement*
- *java.sql.Savepoint*
- *java.sql.Statement*

java.sql.Connection

Table1. JDBC 3.0 Connection Methods Supported

Returns	Signature	Implementation Notes
Savepoint	*setSavepoint (String name)*	Creates a savepoint with the given name in the current transaction and returns the new Savepoint object that represents it.
Savepoint	*setSavepoint ()*	Creates an unnamed savepoint in the current transaction and returns the new Savepoint object that represents it.
void	*releaseSavepoint (Savepoint savepoint)*	Removes the given Savepoint object from the current transaction.
void	*rollback(Savepoint savepoint)*	Undoes all changes made after the given Savepoint object was set.

java.sql.DatabaseMetaData

Table1. JDBC 3.0 DatabaseMetaData Methods Supported

Returns	Signature	Implementation Notes
boolean	*supportsSavepoints()*	'
int	*getDatabaseMajorVersion()*	'
int	*getDatabaseMinorVersion()*	'
int	*getJDBCMajorVersion()*	'
int	*getJDBCMinorVersion()*	'
int	*getSQLStateType()*	'
boolean	*supportsNamedParameters()*	'
boolean	*supportsMultipleOpenResults()*	'
boolean	*supportsGetGeneratedKeys()*	'
boolean	*supportsResultSetHoldability(int holdability)*	'
int	*getResultSetHoldability()*	returns *ResultSet.HOLD_CURSORS_OVER_COMMIT*

java.sql.ParameterMetaData

ParameterMetaData is new in JDBC 3.0. It describes the number, type, and properties of parameters to prepared statements. The method *PreparedStatement.getParameterMetaData* returns a ParameterMetaData object that describes the parameter markers that appear in the *PreparedStatement* object. See *java.sql.PreparedStatement* for more information.

Interface *ParameterMetaData* methods are listed below.

Table1. JDBC 3.0 ParameterMetaData Methods

Returns	Signature	Implementation Notes
int	*getParameterCount()*	'
int	*isNullable(int param)*	'
boolean	*isSigned(int param)*	'
int	*getPrecision(int param)*	'
int	*getScale(int param)*	'
int	*getParameterType(int param)*	'
String	*getParamterTypeName (int param)*	'
String	*getParamterClassName (int param)*	'
int	*getParameterMode (int param)*	'

java.sql.PreparedStatement

The method *PreparedStatement.getParameterMetaData* returns a *ParameterMetaData* object describing the parameter markers that appear in the *PreparedStatement* object. See *java.sql.ParameterMetaData* for more information.

Table1. JDBC 3.0 PreparedStatement Methods

Returns	Signature	Implementation Notes
ParameterMetaData	*getParameterMetaData()*	'

java.sql.Savepoint

The *Savepoint* interface is new in JDBC 3.0. It contains new methods to set, release, or roll back a transaction to designated savepoints. Once a savepoint has been set, the transaction can be rolled back to that savepoint without affecting preceding work. Savepoints provide finer-grained control of transactions by marking intermediate points within a transaction.

Setting and rolling back to a savepoint

The JDBC 3.0 API adds the method *Connection.setSavepoint*, which sets a savepoint within the current transaction. The *Connection.rollback* method has been overloaded to take a savepoint argument. See *java.sql.Connection* for more information.

The code example below inserts a row into a table, sets the savepoint `svpt1`, and then inserts a second row. When the transaction is later rolled back to `svpt1`, the second insertion is undone, but the first insertion remains intact. In other words, when the transaction is committed, only the row containing '1' will be added to TABLE1.

```
conn.setAutoCommit(false); // Autocommit must be off to use savepoints.
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TABLE1 (COL1) VALUES(1)");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("S1");
rows = stmt.executeUpdate("INSERT INTO TABLE1 (COL1) VALUES (2)");
...
conn.rollback(svpt1);
...
conn.commit();
```

Releasing a savepoint

The method *Connection.releaseSavepoint* takes a Savepoint object as a parameter and removes it from the current transaction. Once a savepoint has been released, attempting to reference it in a rollback operation will cause an *SQLException* to be thrown.

Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed or when the entire transaction is rolled back.

Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints created after the savepoint in question.

Rules for savepoints

The savepoint cannot be set within a batch of statements to enabled partial recovery. If a savepoint is set any time before the method *executeBatch* is called, it is set before any of the statements that have been added to the batch are executed.

A savepoint-Name can be reused after it has been released explicitly (by issuing a release of savepoint) or implicitly (by issuing a connection commit/rollback).

Restrictions on savepoints

Derby does not support savepoints within a trigger.

Derby does not release locks as part of the rollback to savepoint.

Table1. JDBC 3.0 Savepoint Methods

Returns	Signature	Implementation Notes
int	*getSavepointId()*	Throws *SQLException* if this is a named savepoint. Retrieves the generated ID for the savepoint that this Savepoint object represents.
String	*getSavepointName()*	Throws *SQLException* if this is an unnamed savepoint. Retrieves the name of the savepoint that this Savepoint object represents.

java.sql.Statement

Table1. JDBC 3.0 Statement Methods

Returns	Signature	Implementation Notes
ResultSet	*getGeneratedKeys()*	'

Autogenerated keys

JDBC 3.0's autogenerated keys feature provides a way to retrieve values from columns that are part of an index or have a default value assigned. Derby supports the autoincrement feature, which allows users to create columns in tables for which the database system automatically assigns increasing integer values. In JDBC 3.0, the method *Statement.getGeneratedKeys* can be called to retrieve the value of such a column. This method returns a *ResultSet* object with a column for the automatically generated key. Calling *ResultSet.getMetaData* on the *ResultSet* object returned by *getGeneratedKeys* produces a *ResultSetMetaData* object that is similar to that returned by IDENTITY_VAL_LOCAL . A flag indicating that any auto-generated columns should be returned is passed to the methods *execute*, *executeUpdate*, or *prepareStatement* when the statement is executed or prepared.

Here's an example that returns a ResultSet with values for auto-generated columns in TABLE1:

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TABLE1 (C11, C12) VALUES
(1,1)",
  Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
```

To use Autogenerated Keys in INSERT statements, pass the `Statement.RETURN_GENERATED_KEYS` flag to the *execute* or *executeUpdate* method. Derby does not support passing column names or column indexes to the *execute*, *executeUpdate*, or *prepareStatement* methods.

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
 }
```

Derby supports the following JDBC escape keywords, which are case-insensitive:

- *JDBC escape keyword for call statements*

  The escape keyword for use in *CallableStatements*.

- *JDBC escape syntax*

  The escape keyword for date formats.

- *JDBC escape syntax for LIKE clauses*

  The keyword for specifying escape characters for LIKE clauses.

- *JDBC escape syntax for fn keyword*

  The escape keyword for scalar functions.

- *JDBC escape syntax for outer joins*

  The escape keyword for outer joins.

- *JDBC espace syntax for time formats*

  The escape keyword for time formats.

- *JDBC escape syntax for timestamp formats*

  The escape keyword for timestamp formats.

Other JDBC escape keywords are not supported.
Note: Derby returns the SQL unchanged in the *Connection.nativeSQL* call, since the escape syntax is native to SQL. In addition, it is unnecessary to call *Statement.setEscapeProcessing* for this reason.

JDBC escape keyword for call statements

This syntax is supported for a *java.sql.Statement* and a *java.sql.PreparedStatement* in addition to a *CallableStatement*.

Syntax

```
{call statement }
```

```
-- Call a Java procedure
{ call TOURS.BOOK_TOUR(?, ?) }
```

JDBC escape syntax

Derby interprets the JDBC escape syntax for date as equivalent to the SQL syntax for

dates.

Syntax

```
{d 'yyyy-mm-dd'}
```

Equivalent to

```
DATE('yyyy-mm-dd')
```

```
VALUES {d '1999-01-09'}
```

JDBC escape syntax for LIKE clauses

The percent sign % and underscore _ are metacharacters within SQL LIKE clauses. JDBC provides syntax to force these characters to be interpreted literally. The JDBC clause immediately following a LIKE expression allows you to specify an escape character:

Syntax

```
WHERE CharacterExpression [ NOT ] LIKE
    CharacterExpressionWithWildCard
    { ESCAPE 'escapeCharacter' }
```

```
-- find all rows in which a begins with the character "%"
SELECT a FROM tabA WHERE a LIKE '$%%' {escape '$'}
-- find all rows in which a ends with the character "_"
SELECT a FROM tabA WHERE a LIKE '%=_' {escape '='}
```

Note: ? is not permitted as an escape character if the LIKE pattern is also a dynamic parameter (?).

In some languages, a single character consists of more than one collation unit (a 16-bit character). The *escapeCharacter* used in the escape clause must be a single collation unit in order to work properly.

You can also use the escape character sequence for LIKE without using JDBC's curly braces; see Boolean expression .

JDBC escape syntax for fn keyword

The fn keyword allows the use of several scalar functions. The function name follows the keyword fn.

Syntax

```
{fn functionCall}
```

where *functionCall* is one of the following functions:

```
concat (
CharacterExpression
,
```

```
CharacterExpression
)
```

Character string formed by appending the second string to the first; if either string is null, the result is NULL. {fn concat (*CharacterExpression*, *CharacterExpression*) is equivalent to built-in syntax { CharacterExpression || CharacterExpression }. For more details, see Concatenation .

```
sqrt (
FloatingPointExpression
)
```

Square root of floating point number.

{fn sqrt (*FloatingPointExpression*)} is equivalent to built-in syntax SQRT(FloatingPointExpression) . For more details see SQRT .

```
abs (
NumericExpression
)
```

Absolute value of number. {fn abs(*NumericExpression*)} is equivalent to built-in syntax ABSOLUTE(NumericExpression) . For more details see ABS or ABSVAL .

```
locate(
CharacterExpression
,
CharacterExpression
  [,
startIndex
] )
```

Position in the second *CharacterExpression* of the first occurrence of the first *CharacterExpression*, searching from the beginning of the second character expression, unless *startIndex* is specified. {fn locate(*CharacterExpression*,*CharacterExpression* [, *startIndex*])} is equivalent to the built-in syntax LOCATE(CharacterExpression, CharacterExpression [, StartPosition]) . For more details see LOCATE .

```
substring(
CharacterExpression
,
startIndex
,
length
)
```

A character string formed by extracting *length* characters from the CharacterExpression beginning at *startIndex*; the index starts with 1.

```
mod(
integer_type
,
integer_type
)
```

MOD returns the remainder (modulus) of argument 1 divided by argument 2. The result is negative only if argument 1 is negative. For more details, see MOD .

Note: Any Derby built-in function is allowed in this syntax, not just those listed in this section.

```
TIMESTAMPADD( interval, integerExpression, timestampExpression )
```

Use the TIMESTAMPADD function to add the value of an interval to a timestamp. The function applies the integer to the specified timestamp based on the interval type and returns the sum as a new timestamp. You can subtract from the timestamp by using negative integers.

Note that TIMESTAMPADD is a JDBC escaped function, and is only accessible using the JDBC escape function syntax.

To perform TIMESTAMPADD on dates and times, it is necessary to convert them to timestamps. Dates are converted to timestamps by putting 00:00:00.0 in the time-of-day fields. Times are converted to timestamps by putting the current date in the date fields.

Note that you should not put a datetime column inside a timestamp arithmetic function in WHERE clauses because the optimizer will not use any index on the column.

```
TIMESTAMPDIFF( interval, timestampExpression1, timestampExpression2 )
```

Use the TIMESTAMPDIFF function to find the difference between two timestamp values at a specified interval. For example, the function can return the number of minutes between two specified timestamps.

Note that TIMESTAMPDIFF is a JDBC escaped function, and is only accessible using the JDBC escape function syntax.

To perform TIMESTAMPDIFF on dates and times, it is necessary to convert them to timestamps. Dates are converted to timestamps by putting 00:00:00.0 in the time-of-day fields. Times are converted to timestamps by putting the current date in the date fields.

Note that you should not put a datetime column inside a timestamp arithmetic function in WHERE clauses because the optimizer will not use any index on the column.

Valid intervals for TIMESTAMPADD and TIMESTAMPDIFF

The TIMESTAMPADD and TIMESTAMPDIFF functions can be used to perform arithmetic with timestamps. These two functions use the following valid intervals for arithmetic operations:

- SQL_TSI_DAY
- SQL_TSI_FRAC_SECOND
- SQL_TSI_HOUR
- SQL_TSI_MINUTE
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_SECOND
- SQL_TSI_WEEK
- SQL_TSI_YEAR

Examples of TIMESTAMPADD and TIMESTAMPDIFF

```
{fn TIMESTAMPADD( SQL_TSI_MONTH, 1, CURRENT_TIMESTAMP)}
```

Returns a timestamp value one month later than the current timestamp.

```
{fn TIMESTAMPDIFF(SQL_TSI_WEEK, CURRENT_TIMESTAMP,
  timestamp('2001-01-01-12.00.00.000000'))}
```

Returns the number of weeks between now and the specified time on January 1, 2001.

JDBC escape syntax for outer joins

Derby interprets the JDBC escape syntax for outer joins (and all join operations) as equivalent to the correct SQL syntax for outer joins or the appropriate join operation.

For information about join operations, see JOIN operation .

Syntax

```
{oj
JOIN operations
 [
JOIN operations
 ]* }
```

Equivalent to

```
JOIN operations
 [
JOIN operations
 ]*
```

```
-- 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 E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM {oj EMPLOYEE E INNER JOIN DEPARTMENT
INNER JOIN EMPLOYEE M ON MGRNO = M.EMPNO ON E.WORKDEPT = DEPTNO};
```

JDBC espace syntax for time formats

Derby interprets the JDBC escape syntax for time as equivalent to the correct SQL syntax for times. Derby also supports the ISO format of 8 characters (6 digits, and 2 decimal points).

Syntax

```
{t 'hh:mm:ss'}
```

Equivalent to

```
TIME 'hh:mm:ss'
```

```
VALUES {t '20:00:03'}
```

JDBC escape syntax for timestamp formats

Derby interprets the JDBC escape syntax for timestamp as equivalent to the correct SQL syntax for timestamps. Derby also supports the ISO format of 23 characters (17 digits, 3 dashes, and 3 decimal points).

Syntax

```
{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

Equivalent to

```
TIMESTAMP 'yyyy-mm-dd hh:mm:ss.f...'
```

The fractional portion of timestamp constants (.f...) can be omitted.

```
VALUES {ts '1999-01-09 20:11:11.123455'}
```

Setting attributes for the database connection URL

Derby allows you to supply a list of attributes to its database connection URL, which is a JDBC feature.

The attributes are specific to Derby.

You typically set attributes in a semicolon-separated list following the protocol and subprotocol. For information on how you set attributes, see Attributes of the Derby database connection URL . This chapter provides reference information only.

Note: Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored.

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.

```
-- boot an encrypted database
jdbc:derby:encryptedDB;bootPassword=cseveryPlace
-- create a new, encrypted database
jdbc:derby:newDB;create=true;dataEncryption=true;
    bootPassword=cseveryPlace
```

create=true

Function

Creates the standard database specified within the database connection URL Derby 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 attribute with other attributes. To specify a territory when creating a database, use the *territory=ll_CC* attribute.
Note: If you specify *create=true* and the database already exists, an *SQLWarning* is

raised.

```
jdbc:derby:sampleDB;create=true
jdbc:derby:;databaseName=newDB;create=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:derby:toursDB*
- *jdbc:derby:;databaseName=toursDB*
- *jdbc:derby:*(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:derby: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 Derby. For more information, see *java.sql.Driver.getPropertyInfo* .

Combining with other attributes

You can combine this attribute with all other attributes.

```
jdbc:derby:;databaseName=newDB;create=true
```

dataEncryption=true

Function

Specifies data encryption on disk for a new database. (For information about data encryption, see "Encrypting Databases on Disk" in the *Derby Developer's Guide* .)

Combining with other attributes

Must be combined with *create=true* and *bootPassword=key* . You have the option of also specifying *encryptionProvider=providerName* and *encryptionAlgorithm=algorithm* .

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    bootPassword=cLo4u922sc23aPe
```

encryptionProvider=providerName

Function

Specifies the provider for data encryption. (For information about data encryption, see "Encrypting Databases on Disk" in the *Derby Developer's Guide* .)

If this attribute is not specified, the default encryption provider is the one included in the jvm that you are using.

Combining with other attributes

Must be combined with *create=true* , *bootPassword=key* , *and dataEncryption=true* . You have the option of also specifying *encryptionAlgorithm=algorithm* .

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
```

encryptionAlgorithm=algorithm

Function

Specifies the algorithm for data encryption.

Specify the algorithm per the Java conventions:

```
algorithmName
/
feedbackMode
/
padding
```

The only padding type allowed with Derby is *NoPadding*.

If no encryption algorithm is specified, the default value is *DES/CBC/NoPadding*.

(For information about data encryption, see "Encrypting Databases on Disk" in Chapter 7 of the *Derby Developer's Guide*).

Combining with other attributes

Must be combined with *create=true* , *bootPassword=key* , *dataEncryption=true* , and *encryptionProvider=providerName* .

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
```

Note: If the specified provider does not support the specified algorithm, Derby throws an exception.

territory=ll_CC

Function

When creating or upgrading a database, use this attribute to associate a non-default territory with the database. Setting the *territory* attribute overrides the default system territory for that database. The default system territory is found using *java.util.Locale.getDefault()*.

Specify a territory in the form *ll_CC*, where *ll* is the two letter language code, and *CC* is

the two letter country code.

Language codes consist of a pair of lower case letters that conform to ISO-639.
Table1. Sample Language Codes

Language Code	Description
de	German
en	English
es	Spanish
ja	Japanese

To see a full list of ISO-639 codes, go to *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt* .

Country codes consist of two uppercase letters that conform to ISO-3166.
Table1. Sample Country Codes

Country Code	Description
DE	Germany
US	United States
ES	Spain
MX	Mexico
JP	Japan

A copy of ISO-3166 can be found at *http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html* .

Combining with other attributes

The *territory* attribute is used only when creating a database.

In the following example, the new database has a territory of Spanish language and Mexican nationality.

```
jdbc:derby:MexicanDB;create=true;territory=es_MX
```

logDevice=logDirectoryPath

Function

The *logDirectoryPath* specifies the path to the directory on which to store the database log during database creation or restore. 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* , *createFrom, restoreFrom, or rollFowardRecoveryFrom*.

```
jdbc:derby:newDB;create=true;logDevice=d:/newDBlog
```

password=userPassword

Function

A valid password for the given user name.

Combining with other attributes

Use in conjunction with *user=userName* .

```
jdbc:derby:toursDB;user=jack;password=upTheHill
```

rollForwardRecoveryFrom=Path

Function

You can specify the *rollForwardRecoveryFrom=Path* in the boot time URL to restore the database using a backup copy and perform rollforward recovery using archived and active logs.

To restore a database using rollforward recovery, you must already have a backup copy of the database, all the archived logs since then, and the active log files. All the log files should be in the database log directory.

After a database is restored from full backup, transactions from the online archived logs and the active logs are replayed.

Combining with other attributes

Do not combine this attribute with createFrom, restoreFrom, or create.

```
URL: jdbc:derby:wombat;rollForwardRecoveryFrom=d:/backup/wombat
```

createFrom=Path

Function

You can specify the *createFrom=Path* attribute in the boot time connection URL to create a database using a full backup at a specified location. If there is a database with the same name in derby.system.home, an error will occur and the existing database will be left intact. If there is not an existing database with the same name in the current derby.system.home location, the whole database is copied from the backup location to the derby.system.home location and started.

The Log files are copied to the default location. The logDevice attribute can be used in conjunction with *createFrom=Path* to store logs in a different location. With *createFrom=Path* you do not need to copy the individual log files to the log directory.

Combining with other attributes

Do not combine this attribute with rollforwardrecoveryFrom, restoreFrom, or create.

```
URL: jdbc:derby:wombat;createFrom=d:/backup/wombat
```

restoreFrom=Path

Function

You can specify the *restoreFrom=Path* attribute in the boot time connection URL to restore a database using a full backup from the specified location. If a database with the same name exists in the derby.system.home location, the whole database is deleted, copied from the backup location, and then restarted.

The log files are copied to the same location they were in when the backup was taken. The logDevice attribute can be used in conjunction with*restoreFrom=Path* to store logs in a different location.

Combining with other attributes

Do not combine this attribute with createFrom, rollforwardrecoveryFrom, or create.

```
URL: jdbc:derby:wombat;restoreFrom=d:/backup/wombat
```

shutdown=true

Function

Shuts down the specified database if you specify a *databaseName*. (Reconnecting to the database reboots the database.)

Shuts down the entire Derby system if and only if you do not specify a *databaseName*

When you are shutting down a single database, it lets Derby perform a final checkpoint on the database.

When you are shutting down a system, it lets Derby 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 Derby has shut down and that there is no connection. Once Derby is shut down, you can restart it by reloading the driver. For details on restarting Derby, see "Shutting Down the System" in Chapter 1 of the *Derby 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.
Note: Any request to the *DriverManager* with a *shutdown=true* attribute raises an exception.

```
-- shuts down entire system
jdbc:derby:;shutdown=true
-- shuts down salesDB
jdbc:derby:salesDB;shutdown=true
```

user=userName

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* .

The following database connection URL connects the user jill to *toursDB*:

```
jdbc:derby:toursDB;user=jill;password=toFetchAPail
```

(no attributes)

If no attributes are specified, you must specify a *databaseName.*

Derby 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.

```
jdbc:derby:mydb
```

J2EE Compliance: Java Transaction API and javax.sql Extensions

J2EE, or the Java 2 Platform, Enterprise Edition, is a standard for development of enterprise applications based on reusable components in a multi-tier environment. In addition to the features of the Java 2 Platform, Standard Edition (J2SE) J2EE adds support for Enterprise Java Beans (EJBs), Java Server Pages (JSPs), Servlets, XML and many more. The J2EE architecture is used to bring together existing technologies and enterprise applications in a single, manageable environment.

Derby 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. Within this system, Derby can serve as the resource manager.

For more information on J2EE, see the J2EE specification available at *http://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 (Derby). A user requesting a connection can get one of the available connections 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, making the performance advantage of a connection pool negligible. 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.

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 Derby as a resource manager, see Chapter 6, "Using Derby as a J2EE Resource Manager" in the *Derby Developer's Guide* .

JVM and libraries for J2EE features

These features require the following:

- Java 2 Platform, Standard Edition v 1.2 (J2SE) environment or greater
- *javax.sql* libraries

  The JDBC 2.0 standard extension binaries are available from *http://java.sun.com/products/jdbc/download.html* . These libraries are part of the standard environment from Java 2 Platform, Standard Edition v 1.4 or later.

- *javax.transaction.xa* libraries

  These libraries are part of the standard environment from Java 2 Platform, Standard Edition v 1.4 or later.

  For the JTA libraries, see *http://java.sun.com/products/jta/* and download the specification and javadoc help files for JTA interfaces.

- Derby (*derby.jar*)

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. Derby fully implements this API.

- *javax.transaction.xa.XAResource*
- *javax.transaction.xa.Xid*
- *javax.transaction.xa.XAException*

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, 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 roll back that in-doubt transaction.
Note: Use the network client driver's XA DataSource interface (org.apache.derby.jdbc.ClientXADataSource) when XA support is required in a remote (client/server) environment.

javax.sql: JDBC Extensions

This section documents the JDBC extensions that Derby implements for J2EE compliance. (For more details about these extensions, see *http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/javax/sql/package-summary.html*).

- *javax.sql.DataSource*

  Derby'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.

- *javax.sql.ConnectionPoolDataSource and javax.sql.PooledConnection*

  Establishing a connection to the database can be a relatively expensive operation in client/server environments. Establishing the connection once and then using the same connection for multiple requests can dramatically improve the performance of a database.

  The Derby implementation of `ConnectionPoolDataSource` and `PooledConnection` allows a connection pool server to maintain a set of such connections to the resource manager (Derby). In an embedded environment, connections are much cheaper and connection pooling is not necessary.

- *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 *XAConnections*.

In addition, Derby provides three methods for *XADataSource, DataSource, and ConnectionPoolDataSource*. Derby supports a number of additional data source properties:

- *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".

Note: Set these properties before getting the connection.

Derby API

Derby provides Javadoc HTML files of API classes and interfaces in the *javadoc* subdirectory.

This appendix provides a brief overview of the API. Derby does not provide the Javadoc for the *java.sql* packages, the main API for working with Derby, because it is included in the JDBC API. For information about Derby's implementation of JDBC, see JDBC Reference .

This document divides the API classes and interfaces into several categories. The stand-alone tools and utilities are java classes that stand on their own and are invoked in a command window. The JDBC implementation classes are standard JDBC APIs, and are not invoked on the command-line. Instead, you invoke these only within a specified context within another application.

Stand-alone tools and utilities

These classes are in the packages *org.apache.derby.tools*.

- *org.apache.derby.tools.ij*

  An SQL scripting tool that can run as an embedded or a remote client/server application. See the *Derby Tools and Utilities Guide* .

- *org.apache.derby.tools.sysinfo*

  A command-line, server-side utility that displays information about your JVM and Derby product. See the *Derby Tools and Utilities Guide* .

- *org.apache.derby.tools.dblook*

  A utility to view all or parts of the Data Definition Language (DDL) for a given database. See the *Derby Tools and Utilities Guide* .

JDBC implementation classes

JDBC driver

This is the JDBC driver for Derby:

- *org.apache.derby.jdbc.EmbeddedDriver*

  Used to boot the embedded built-in JDBC driver and the Derby system.

- *org.apache.derby.jdbc.ClientDriver*

  Used to connect to the Derby Network Server in client-server mode.

See the *Derby Developer's Guide* .

Data Source Classes

These classes are all related to Derby's implementation of *javax.sql.DataSource* and related APIs. For more information, see the *Derby Developer's Guide* .

Embedded environment:

- *org.apache.derby.jdbc.EmbeddedDataSource*
- *org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource*
- *org.apache.derby.jdbc.EmbeddedXADataSource*

Client-server environment

- *org.apache.derby.jdbc.ClientDataSource*
- *org.apache.derby.jdbc.ClientConnectionPoolDataSource*
- *org.apache.derby.jdbc.ClientXADataSource*

Miscellaneous utilities and interfaces

- *org.apache.derby.authentication.UserAuthenticator*
- An interface provided by Derby. Classes that provide an alternate user authentication scheme must implement this interface. For information about users, see "Working with User Authentication" in Chapter 7 of the *Derby Developer's Guide* .

Supported territories

The following is a list of supported territories:

Territory	Derby territory setting (derby.territory)
Japanese	ja_JP
Korean	ko_KR
Chinese (Traditional)	zh_TW
Chinese (Simplified)	zh_CN
French	fr
German	de_DE
Italian	it
Spanish	es
Portuguese (Brazilian)	pt_BR

Derby limitations

The section lists the limitations associated with Derby.

Limitations on identifier length

Table1. Identifier length limitations

The following table lists limitations on identifier lengths in Derby.

Identifier	Maximum number of characters allowed
constraint name	128
correlation name	128
cursor name	128
data source column name	128
data source index name	128
data source name	128
savepoint name	128
schema name	128
unqualified column name	128
unqualified function name	128
unqualified index name	128
unqualified procedure name	128
parameter name	128
unqualified trigger name	128
unqualified table name, view name, stored procedure name	128

Numeric limitations

Table1. Numeric limitations

The following table contains limitations on numeric values in Derby.

Value	Limit
Smallest INTEGER	-2,147,483,648
Largest INTEGER	2,147,483,647
Smallest BIGINT	-9,223,372,036,854,775,808
Largest BIGINT	9,223,372,036,854,775,807
Smallest SMALLINT	-32,768
Largest SMALLINT	32,767
Largest decimal precision	31,255
Smallest DOUBLE	-1.79769E+308
Largest DOUBLE	1.79769E+308
Smallest positive DOUBLE	2.225E-307
Largest negative DOUBLE	-2.225E-307

Value	Limit
Smallest REAL	-3.402E+38
Largest REAL	3.402E+38
Smallest positive REAL	1.175E-37
Largest negative REAL	-1.175E-37

String limitations

Table1. String limitations

The following table contains limitations on string values in Derby.

Value	Maximum Limit
Length of CHAR	254 characters
Length of VARCHAR	32,672 characters
Length of LONG VARCHAR	32,700 characters
Length of CLOB	2,147,483,647 characters
Length of BLOB	2,147,483,647 characters
Length of character constant	32,672
Length of concatenated character string	2,147,483,647
Length of concatenated binary string	2,147,483,647
Number of hex constant digits	16,336
Length of DOUBLE value constant	30 characters

DATE, TIME, and TIMESTAMP limitations

Table1. DATE, TIME, and TIMESTAMP limitations

The following table lists limitations on date, time, and timestamp values in Derby.

Value	Limit
Smallest DATE value	0001-01-01
Largest DATE value	9999-12-31
Smallest TIME value	00:00:00
Largest TIME value	24:00:00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000
Largest TIMESTAMP value	9999-12-31-24.00.00.000000

Limitations for database manager values

Table1. Database manager limitations

The following table lists limitations on various Database Manager values in Derby.

Value	Limit
Maximum columns in a table	1,012
Maximum columns in a view	5,000
Maximum number of parameters in a stored procedure	90

Value	Limit
Maximum indexes on a table	32,767 or storage capacity
Maximum tables referenced in an SQL statement or a view	storage capacity
Maximum elements in a select list	1,012
Maximum predicates in a WHERE or HAVING clause	storage capacity
Maximum number of columns in a GROUP BY clause	32,677
Maximum number of columns in an ORDER BY clause	1,012
Maximum number of prepared statements	storage capacity
Maximum declared cursors in a program	storage capacity
Maximum number of cursors opened at one time	storage capacity
Maximum number of constraints on a table	storage capacity
Maximum level of subquery nesting	storage capacity
Maximum number of subqueries in a single statement	storage capacity
Maximum number of rows changed in a unit of work	storage capacity
Maximum constants in a statement	storage capacity
Maximum depth of cascaded triggers	16

Trademarks

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the Apache Derby documentation library:

Cloudscape, DB2, DB2 Universal Database, DRDA, and IBM are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.