Derby Reference Manual

Version 10.13

Derby Document build:
November 9, 2016, 4:47:32 AM (UTC)

Contents

Copyright

Related information

License

License

The Apache License, Version 2.0

```
      that are managed by, or on behalf of, the Licensor for the
      purpose of discussing and improving the Work, but excluding
      communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a
      Contribution."

      "Contributor" shall mean Licensor and any individual or Legal
      Entity on behalf of whom a Contribution has been received by
      Licensor and subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions
      of this License, each Contributor hereby grants to You a
      perpetual, worldwide, non-exclusive, no-charge, royalty-free,
      irrevocable copyright license to reproduce, prepare Derivative
      Works of, publicly display, publicly perform, sublicense, and
      distribute the Work and such Derivative Works in Source or
      Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have
      made, use, offer to sell, sell, import, and otherwise transfer
      the Work, where such license applies only to those patent claims
      licensable by such Contributor that are necessarily infringed by
      their Contribution(s) alone or by combination of their
      Contribution(s) with the Work to which such Contribution(s) was
      submitted. If You institute patent litigation against any entity
      (including a cross-claim or counterclaim in a lawsuit) alleging
      that the Work or a Contribution incorporated within the Work
      constitutes direct or contributory patent infringement, then any
      patent licenses granted to You under this License for that Work
      shall terminate as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent notices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative Works
          that You distribute, all copyright, patent, trademark, and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any part of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of its
          distribution, then any Derivative Works that You distribute
          must include a readable copy of the attribution notices
          contained within such NOTICE file, excluding those notices
          that do not pertain to any part of the Derivative Works, in
          at least one of the following places: within a NOTICE text
          file distributed as part of the Derivative Works; within the
          Source form or documentation, if provided along with the
          Derivative Works; or, within a display generated by the
          Derivative Works, if and wherever such third-party notices
          normally appear. The contents of the NOTICE file are for
          informational purposes only and do not modify the License.
          You may add Your own attribution notices within Derivative
          Works that You distribute, alongside or as an addendum to
          the NOTICE text from the Work, provided that such additional
          attribution notices cannot be construed as modifying the
          License.

      You may add Your own copyright statement to Your modifications
```

and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.  See the License for the specific language governing
permissions and limitations under the License.
```

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*.

For more information about Derby, visit the Derby website at http://db.apache.org/derby. The website provides pointers to the Derby Wiki and other resources, such as the derby-users mailing list, where you can ask questions about issues not covered in the documentation.

Purpose of this document

This document, 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 document 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 documentation 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.
- *Argument matching*

  Information about argument matching in Java functions and procedures.
- *SQL reserved words*

  SQL keywords beyond the standard keywords.
- *Derby support for SQL:2011 features*

  Lists of SQL:2011 features that Derby does and does not support.
- *Derby system tables*

  Reference information about the Derby system catalogs.
- *XPLAIN style tables*

  Information about the optional XPLAIN style system tables.
- *Derby exception messages and SQL states*

  Information about Derby exception messages.
- *JDBC reference*

  Information about Derby's implementation of the Java Database Connectivity (JDBC) API.
- *Setting attributes for the database connection URL*

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

- *Derby property reference*

  Information about Derby properties.
- *Java EE compliance: Java Transaction API and javax.sql interfaces*

  Information about Derby's support for the Java EE platform, in particular support for the Java Transaction API and the JDBC API.
- *Derby API*

  Notes about proprietary APIs for Derby.
- *Supported locales*

  Locales supported by Derby.
- *Derby limitations*

  Limitations of Derby.

SQL syntax used in this manual

SQL syntax is presented in modified BNF notation.

The meta-symbols of BNF are listed in the following table.

Table 1. BNF meta-symbols

Symbol	Meaning
|	Or. Choose one of the items.
[ ]	Encloses optional items.
*	Flags items that you can repeat 0 or more times. Has a special meaning in some SQL statements.
{ }	Groups required items so that they can be marked with the symbol |. Has a special meaning in JDBC escape syntax (see JDBC escape syntax).
( ) . ,	Other punctuation that is part of the syntax.

The following example shows how SQL syntax is presented:

```
CREATE [ UNIQUE ] INDEX indexName
  ON tableName ( simpleColumnName [ , simpleColumnName ]* )
```

SQL language reference

Derby implements a subset of the SQL standard.

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 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-style identifiers are case-insensitive (see *SQLIdentifier*), 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 expressions.
- Comments can be either single-line or multiline as per the SQL standard. Single-line comments start with two dashes (--) and end with the newline character. Multiline comments are bracketed, start with forward slash star (/*), and end with star forward slash (*/). Note that bracketed comments may be nested. Any text between the starting and ending comment character sequence is ignored.

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 identifier rules and are thus called *SQLIdentifier*s.

Rules for SQL 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").

SQLIdentifier

An *SQLIdentifier* is a dictionary object identifier that conforms to the rules of SQL. SQL 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 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 *SQLIdentifiers* 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 *SQLIdentifier* is "dot-separated." You qualify a dictionary object name in order to avoid ambiguity.

aggregateName

An *aggregateName* represents a user-defined aggregate (UDA). To create a UDA, use the CREATE DERBY AGGREGATE statement.

Syntax

```
[ schemaName. ] SQLIdentifier
```

You can qualify an aggregate name with a *schemaName*. If a qualified aggregate name is specified, the schema name cannot begin with SYS.

Example

```
-- types.maxPrice is an aggregateName that includes a schemaName
CREATE DERBY AGGREGATE types.maxPrice FOR PRICE
EXTERNAL NAME 'com.example.myapp.types.PriceMaxer';
```

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 "Users and authorization identifiers" in the *Derby Security Guide*.

Syntax

```
SQLIdentifier
```

Example

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

columnName

In many places in the SQL syntax, you can represent the name of a column by qualifying it with a *tableName* or *correlationName*.

In some situations, you cannot qualify a *columnName* with a *tableName* or a *correlationName*, but must use a *simpleColumnName* instead. Those situations are:

- Creating a table (CREATE TABLE statement)
- Specifying updatable columns in a cursor
- In a column's correlation name in a SELECT expression (see *selectExpression*)
- In a column's correlation name in a *tableExpression* (see *tableExpression*)

You cannot use *correlationName*s for updatable columns; using *correlationName*s 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 *correlationName*`col1 FOR c11` is not permitted because `c11` is listed in the FOR UPDATE list of columns. You can use the *correlationName*`FOR c12` because it is not in the FOR UPDATE list.

Syntax

```
[ { tableName | correlationName } . ] SQLIdentifier
```

Example

```
-- C.Country is a columnName qualified with a correlationName.
SELECT C.Country
FROM APP.Countries C
```

constraintName

A *constraintName* represents a constraint (see CONSTRAINT clause).

Syntax

```
[ schemaName. ] SQLIdentifier
```

You can qualify a *constraintName* with a *schemaName*, but the *schemaName* of the constraint must be the same as the *schemaName* of the table on which the constraint is placed.

Example

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

```
-- s1.c1 is a schema-qualified constraint; the schema name
--   is not required here, but if specified must match that of the table
CREATE SCHEMA s1;
CREATE TABLE s1.t1 ( a INT, CONSTRAINT s1.c1 CHECK ( a > 0 ) );
```

correlationName

A *correlationName* is given to a table expression in a FROM clause as a new name or alias for that table. You cannot qualify a *correlationName* with a *schemaName*.

You cannot use *correlationName*s for updatable columns; using *correlationName*s 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 *correlationName*`col1 FOR c11` is not permitted because `c11` is listed in the FOR UPDATE list of columns. You can use the *correlationName*`FOR c12` because it is not in the FOR UPDATE list.

Syntax

```
SQLIdentifier
```

Example

```
-- C is a correlationName
SELECT C.NAME
FROM SAMP.STAFF C
```

cursorName

A *cursorName* 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 *cursorName*.

Syntax

```
SQLIdentifier
```

Example

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

functionName

A *functionName* represents a Java function. To create a function, use the CREATE FUNCTION statement.

Syntax

```
[ schemaName. ] SQLIdentifier
```

You can qualify a function name with a *schemaName*. If a qualified function name is specified, the schema name cannot begin with SYS.

Example

```
-- Declaring a scalar function
CREATE FUNCTION TO_DEGREES
( RADIANS DOUBLE )
RETURNS DOUBLE
PARAMETER STYLE JAVA
NO SQL LANGUAGE JAVA
EXTERNAL NAME 'java.lang.Math.toDegrees';
```

indexName

An *indexName* represents an index. To create an index, use the CREATE INDEX statement.

Syntax

```
[ schemaName . ] SQLIdentifier
```

You can qualify an index name with a *schemaName*. If a qualified index name is specified, the schema name cannot begin with SYS.

Example

```
DROP INDEX APP.ORIGINDEX;
-- OrigIndex is an indexName without a schemaName
CREATE INDEX ORIGINDEX ON FLIGHTS (ORIG_AIRPORT)
```

newTableName

A *newTableName* represents a renamed table (see RENAME TABLE statement). You cannot qualify a *newTableName* with a *schemaName*.

Syntax

```
SQLIdentifier
```

Example

```
-- FLIGHTAVAILABLE is a newTableName
RENAME TABLE FLIGHTAVAILABILITY TO FLIGHTAVAILABLE
```

procedureName

A *procedureName* represents a Java stored procedure. To create a procedure, use the CREATE PROCEDURE statement.

Syntax

```
[ schemaName. ] SQLIdentifier
```

You can qualify a procedure name with a *schemaName*. If a qualified procedure name is specified, the schema name cannot begin with SYS.

Example

```
-- SALES.TOTAL_REVENUE is a procedureName that includes a schemaName
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.example.sales.calculateRevenueByMonth'
```

roleName

A *roleName* refers to an SQL role. A role in a database is uniquely identified by its role name.

Syntax

```
SQLIdentifier
```

In terms of SQL, a role name is also technically an *authorizationIdentifier*, but that term is often used for user names in Derby for historical reasons.

Example

```
DROP ROLE reader
```

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

```
SQLIdentifier
```

Example

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

sequenceName

A *sequenceName* represents a sequence generator. To create a sequence generator, use the CREATE SEQUENCE statement.

Syntax

```
[ schemaName. ] SQLIdentifier
```

You can qualify a sequence name with a *schemaName*. If a qualified sequence name is specified, the schema name cannot begin with SYS.

Example

```
DROP SEQUENCE order_id RESTRICT
```

simpleColumnName

A *simpleColumnName* is used to represent a column when it cannot be qualified by a *tableName* orb*correlationName*. This is the case when the qualification is fixed, as it is in a column definition within a CREATE TABLE statement.

Syntax

```
SQLIdentifier
```

Example

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

synonymName

A *synonymName* represents a synonym for a table or a view. To create a synonym, use the CREATE SYNONYM statement.

Syntax

```
[ schemaName. ] SQLIdentifier
```

You can qualify a synonym name with a *schemaName*. If a qualified synonym name is specified, the schema name cannot begin with SYS.

tableName

A *tableName* represents a table. To create a table, use the CREATE TABLE statement.

Syntax

```
[ schemaName. ] SQLIdentifier
```

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

Example

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

triggerName

A *triggerName* refers to a trigger created by a user. To create a trigger, use the CREATE TRIGGER statement.

Syntax

```
[ schemaName . ] SQLIdentifier
```

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

Example

```
DROP TRIGGER TRIG1
```

typeName

A *typeName* represents a user-defined type (UDT). To create a UDT, use the CREATE TYPE statement.

Syntax

```
[ schemaName. ] SQLIdentifier
```

You can qualify a type name with a *schemaName*. If a qualified type name is specified, the schema name cannot begin with SYS.

Example

```
CREATE TYPE price
EXTERNAL NAME 'com.example.types.Price'
LANGUAGE JAVA
```

viewName

A *viewName* represents a table or a view. To create a view, use the CREATE VIEW statement.

Syntax

```
[ schemaName. ] SQLIdentifier
```

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

Example

```
-- This is a view qualified by a schemaName
SELECT COUNT(*) FROM SAMP.EMP_RESUME
```

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/View 'MYTABLE' does not exist.
```

ALTER TABLE statement

The ALTER TABLE statement modifies a table.

The ALTER TABLE statement allows you to:

- Add a column to a table
- Add a constraint to a table
- Drop a column from a table
- Drop an existing constraint from a table
- Increase the width of a BLOB, CLOB, VARCHAR, or VARCHAR FOR BIT DATA column
- Override row-level locking for the table (or drop the override)
- Change the increment value and start value of the identity column
- Change an identity column from ALWAYS to DEFAULT BY behavior or vice-versa
- Change the nullability constraint for a column
- Change the default value for a column

Syntax

```
ALTER TABLE tableName
{
    ADD COLUMN columnDefinition |
    ADD CONSTRAINT clause |
    DROP [ COLUMN ] columnName [ CASCADE | RESTRICT ] |
    DROP { PRIMARY KEY |
           FOREIGN KEY constraintName |
           UNIQUE constraintName |
           CHECK constraintName |
           CONSTRAINT constraintName } |
    ALTER [ COLUMN ] columnAlteration |
    LOCKSIZE { ROW | TABLE }
}
```

columnAlteration

```
columnName SET DATA TYPE BLOB( integer
 )
 |
columnName SET DATA TYPE CLOB( integer
 )
 |
columnName SET DATA TYPE VARCHAR( integer
```

```
 )
 |
columnName SET DATA TYPE VARCHAR( integer
 ) FOR BIT DATA
 |
columnName SET INCREMENT BY integerConstant |
columnName RESTART WITH integerConstant |
columnName SET GENERATED { ALWAYS | BY
 DEFAULT }
 |
columnName { SET | DROP } NOT NULL
 |
columnName [ NOT ] NULL
 |
columnName [ WITH | SET ] DEFAULT defaultValue |
columnName DROP DEFAULT
```

In the *columnAlteration*, SET INCREMENT BY *integerConstant* 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.

RESTART WITH *integerConstant* specifies the next value to be generated for the identity column. RESTART WITH is useful for a table that has an identity column that was defined as GENERATED BY DEFAULT and that has a unique key defined on that identity column. Because GENERATED BY DEFAULT allows both manual inserts and system generated values, it is possible that manually inserted values can conflict with system generated values. To work around such conflicts, use the RESTART WITH syntax to specify the next value that will be generated for the identity column. Consider the following example, which involves a combination of automatically generated data and manually inserted data:

```
CREATE TABLE tauto(i INT GENERATED BY DEFAULT AS IDENTITY, k INT)
CREATE UNIQUE INDEX tautoInd ON tauto(i)
INSERT INTO tauto(k) values 1,2
```

The system will automatically generate values for the identity column. But now you need to manually insert some data into the identity column:

```
INSERT INTO tauto VALUES (3,3)
INSERT INTO tauto VALUES (4,4)
INSERT INTO tauto VALUES (5,5)
```

The identity column has used values 1 through 5 at this point. If you now want the system to generate a value, the system will generate a 3, which will result in a unique key exception because the value 3 has already been manually inserted. To compensate for the manual inserts, issue an ALTER TABLE statement for the identity column with RESTART WITH 6:

```
ALTER TABLE tauto ALTER COLUMN i RESTART WITH 6
```

SET GENERATED ALWAYS causes Derby to not accept an overriding value for an identity column when a row is inserted or updated. SET GENERATED BY DEFAULT causes Derby to permit these overrides.

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.

Derby raises an error if you try to change the *DataType* of a generated column to a type which is not assignable from the type of the *generationClause*. Derby also raises an error if you try to add a DEFAULT clause to a generated column.

Adding columns

The syntax for the *columnDefinition* for a new column is the same as for a column in a CREATE TABLE statement. This syntax allows a column constraint to 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 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.

If you add a generated column to a table, Derby computes the generated values for all existing rows in the table.

ALTER TABLE ADD COLUMN adds the new column at the end of the table row. If you need to change a column in a way not permitted by ALTER TABLE ALTER COLUMN (for example, if you need to change its data type), the only way to do so is to drop the column and add a new one, and this changes the ordering of the columns.

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 columns

ALTER TABLE DROP COLUMN allows you to drop a column from a table.

The keyword COLUMN is optional.

The keywords CASCADE and RESTRICT are also optional. If you specify neither CASCADE nor RESTRICT, the default is CASCADE.

If you specify RESTRICT, then the column drop will be rejected if it would cause a dependent schema object to become invalid.

If you specify CASCADE, then the column drop should additionally drop other schema objects which have become invalid.

The schema objects which can cause a DROP COLUMN RESTRICT to be rejected include: views, triggers, primary key constraints, foreign key constraints, unique key constraints, check constraints, and column privileges. If one of these types of objects depends on the column being dropped, DROP COLUMN RESTRICT will reject the statement.

Derby also raises an error if you specify RESTRICT when you drop a column referenced by the *generationClause* of a generated column. However, if you specify CASCADE, the generated column is also dropped with CASCADE semantics.

You may not drop the last (only) column in a table.

CASCADE/RESTRICT doesn't consider whether the column being dropped is used in any indexes. When a column is dropped, it is removed from any indexes which contain it. If that column was the only column in the index, the entire index is dropped.

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 *columnAlteration* allows you to alter the named column in the following ways:

- Increasing the width of an existing VARCHAR or VARCHAR FOR BIT DATA 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 *integerConstant*. 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.
- Modifying the nullability constraint of a column.

  You can add the NOT NULL constraint to an existing column. To do so there must not be existing NULL values for the column in the table.

  You can remove the NOT NULL constraint from an existing column. To do so the column must not be used in a PRIMARY KEY constraint.
- Changing an identity column from GENERATED ALWAYS to GENERATED BY DEFAULT behavior or vice-versa.

  The SET GENERATED clause may only be applied to identity columns. It cannot be used to convert a non-identity column into an identity column. This clause can be useful if you need to preserve key values when bulk-loading a table from a snapshot or exported dump.
- Changing the default value for a column.

  You can use DEFAULT *default-value* to change a column default. To disable a previously set default, use DROP DEFAULT (alternatively, you can specify NULL as the *default-value*).

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 the city_id column if there are no dependent objects:
ALTER TABLE Cities DROP COLUMN city_id RESTRICT;
-- Drop the city_id column, also dropping all dependent objects:
ALTER TABLE Cities DROP COLUMN city_id CASCADE;

-- 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;

-- Remove the NOT NULL constraint from the MANAGER column
ALTER TABLE Employees ALTER COLUMN Manager NULL;
-- Add the NOT NULL constraint to the SSN column
ALTER TABLE Employees ALTER COLUMN ssn NOT NULL;
```

```
-- Change the default value for the SALARY column
ALTER TABLE Employees ALTER COLUMN Salary DEFAULT 1000.0
ALTER TABLE Employees ALTER COLUMN Salary DROP DEFAULT

-- Bulk load a table by temporarily changing a GENERATED ALWAYS identity
 column
-- into a GENERATED BY default column.
-- After loading the table, reset the identity column to be GENERATED
 ALWAYS
-- and move its sequence number forward past the last inserted key.
ALTER TABLE targetTable ALTER COLUMN keyCol SET GENERATED BY DEFAULT;
INSERT INTO targetTable SELECT * FROM sourceTable;
ALTER TABLE targetTable ALTER COLUMN keyCol SET GENERATED ALWAYS;
ALTER TABLE targetTable ALTER COLUMN keyCol RESTART WITH 1234567;
```

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.

CALL (PROCEDURE) statement

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

When a procedure with definer's rights is called, the current default schema is set to the eponymously named schema of the definer. For example, if the defining user is called OWNER, the default schema will also be set to OWNER. The definer's rights include the right to set the current role to a role for which the definer has privileges. When the procedure is first invoked, no role is set; even if the invoker has set a current role, the procedure running with definer's rights has no current role set initially.

When a procedure with invoker's rights is called, the current default schema and current role are unchanged initially within the procedure. Similarly, if SQL authorization mode is not enabled, the current default schema is unchanged initially within the procedure.

When the call returns, any changes made inside the procedure to the default current schema (and current role, if relevant) are reset (popped).

For information about definer's rights, see EXTERNAL SECURITY.

Syntax

```
CALL procedureName ( [ expression [ , expression ]* ] )
```

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.example.sales.calculateRevenueByMonth';
CALL SALES.TOTAL_REVENUE(?,?,?);
```

CREATE statements

Use the CREATE statements to create functions, indexes, procedures, roles, schemas, synonyms, tables, triggers, and views.

CREATE DERBY AGGREGATE statement

The CREATE DERBY AGGREGATE statement creates a user-defined aggregate (UDA). A UDA is a custom aggregate operator.

Syntax

```
CREATE DERBY AGGREGATE aggregateName FOR valueDataType
[ RETURNS returnDataType
 ]
EXTERNAL NAMEsingleQuotedString
```

The aggregate name is composed of an optional *schemaName* and a *SQLIdentifier*. If a *schemaName* is not provided, the current schema is the default schema. If a qualified aggregate name is specified, the schema name cannot begin with SYS.

In general, UDAs live in the same namespace as one-argument user-defined functions (see CREATE FUNCTION statement). A schema-qualified UDA name may not be the schema-qualified name of a one-argument user-defined function.

An unqualified UDA name (that is, the UDA name without its schema name) may not be the name of an aggregate defined in part 2 of the SQL Standard, section 10.9:

```
ANY
AVG
COLLECT
COUNT
EVERY
FUSION
INTERSECTION
MAX
MIN
SOME
STDDEV_POP
STDDEV_SAMP
SUM
VAR_POP
VAR_SAMP
```

In addition, an unqualified UDA name may not be the name of any of the Derby built-in functions which take one argument.

The *valueDataType* can be any valid nullable Derby data type except for XML, including user-defined types.

The *returnDataType* can be any valid nullable Derby data type except for XML. If the *returnDataType* is omitted, it defaults to be the same as *valueDataType*.

The *singleQuotedString* specified by the EXTERNAL NAME clause is the full name of a Java class which implements the *org.apache.derby.agg.Aggregator* interface. That contract is not checked until a statement is compiled which invokes the UDA.

The *org.apache.derby.agg.Aggregator* interface extends *java.io.Serializable*, so you must make sure that all of the state of your UDA is serializable. A UDA may be serialized to disk when it performs grouped aggregation over a large number of groups. That is, intermediate results may be serialized to disk for a query like the following:

```
SELECT a, myAggregate( b ) FROM myTable GROUP BY a
```

The serialization will fail if the UDA contains non-serializable fields.

The owner of the schema where the UDA lives automatically gains the USAGE privilege on the UDA and can grant this privilege to other users and roles. Only the database owner and the owner of the UDA can grant these USAGE privileges. The USAGE privilege cannot be revoked from the schema owner. See GRANT statement and REVOKE statement for more information.

Examples

```
CREATE DERBY AGGREGATE mode FOR INT
EXTERNAL NAME 'com.example.myapp.aggs.Mode';

CREATE DERBY AGGREGATE types.maxPrice FOR PRICE
```

```
EXTERNAL NAME 'com.example.myapp.types.PriceMaxer';

CREATE DERBY AGGREGATE types.avgLength FOR VECTOR
RETURNS DOUBLE
EXTERNAL NAME 'com.example.myapp.types.VectorLength';
```

See "Programming user-defined aggregates" in the *Derby Developer's Guide* for more details about creating and using user-defined aggregates.

CREATE FUNCTION statement

The CREATE FUNCTION statement creates a Java function, which you can then use in an expression.

The function owner and the database owner automatically gain the EXECUTE privilege on the function, and are able to grant this privilege to other users. The EXECUTE privileges cannot be revoked from the function and database owners.

For details on how Derby matches procedures to Java methods, see Argument matching. For information on how functions interact with deferrable constraints, see Deferrable constraints.

Syntax

```
CREATE FUNCTION functionName ( [ functionParameter
   [ , functionParameter ]* [...] ] ) RETURNS returnDataType
[ functionElement ]*
```

An ellipsis (`...`) after the last parameter indicates that the Java method supports trailing optional arguments, called varargs. The ellipsis indicates that the method may be invoked with zero or more trailing values, all having the data type of the last argument.

functionParameter

```
[ parameterName ] dataType
```

A *parameterName* must be unique within a function.

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

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

returnDataType

```
tableType | dataType
```

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

tableType

```
TABLE( columnElement [, columnElement ]* )
```

This is the return type of a table function. Currently, only Derby-style table functions are supported. They are functions which return JDBC *ResultSet*s. For more information, see "Programming Derby-style table functions" in the *Derby Developer's Guide*.

At runtime, as values are read out of the user-supplied *ResultSet*, Derby coerces those values to the data types declared in the CREATE FUNCTION statement. This affects values typed as CHAR, VARCHAR, LONG VARCHAR, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, LONG VARCHAR FOR BIT DATA, and DECIMAL/NUMERIC. Values which are too long are truncated to the maximum length declared in the CREATE FUNCTION statement. In addition, if a *String* value is returned in the *ResultSet* for a column of CHAR type and the *String* is shorter than the declared length of the CHAR column, Derby pads the end of the *String* with blanks in order to stretch it out to the declared length.

columnElement

```
SQLIdentifier dataType
```

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

Note: XML is not allowed as the type of a column in the dataset returned by a table function.

functionElement

```
{
    LANGUAGE JAVA |
    { DETERMINISTIC | NOT DETERMINISTIC } |
    EXTERNAL NAME singleQuotedString |
    PARAMETER STYLE { JAVA | DERBY_JDBC_RESULT_SET | DERBY } |
    EXTERNAL SECURITY { DEFINER | INVOKER } |
    { NO SQL | CONTAINS SQL | READS SQL DATA } |
    { RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT }
}
```

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

LANGUAGE JAVA

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

DETERMINISTIC, NOT DETERMINISTIC

DETERMINISTIC declares that the function is deterministic, meaning that with the same set of input values, it always computes the same result. The default is NOT DETERMINISTIC. Derby cannot recognize whether an operation is actually deterministic, so you must take care to specify this element correctly.

EXTERNAL NAME *singleQuotedString*

The *singleQuotedString* specified by the EXTERNAL NAME clause describes the Java method to be called when the function is executed.

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 can be 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.

DERBY_JDBC_RESULT_SET

The PARAMETER STYLE is DERBY_JDBC_RESULT_SET if and only if this is a Derby-style table function, that is, a function which returns *tableType* and which is mapped to a method which returns a JDBC *ResultSet*.

DERBY

The PARAMETER STYLE must be DERBY if and only if an ellipsis (`...`) appears at the end of the argument list.

EXTERNAL SECURITY

If SQL authorization mode is enabled, a function runs by default with the privileges specified for the user who invokes the function (invoker's rights). To specify that the function should run with the privileges specified for the user who defines the function (definer's rights), create the function with EXTERNAL SECURITY DEFINER. Those privileges include the right to set the current role to a role for which the definer has privileges. When the function is first invoked, no role is set; even if the invoker has set a current role, the function running with definer's rights has no current role set initially.

See *derby.database.sqlAuthorization* for details about setting SQL authorization mode.

When a function with definer's rights is invoked, the current default schema is set to the eponymously named schema of the definer. For example, if the defining user is called OWNER, the default schema will also be set to OWNER.

When a function with invoker's rights is called, the current default schema and current role are unchanged initially within the function. Similarly, if SQL authorization mode is not enabled, the current default schema is unchanged initially within the function.

When the call returns, any changes made inside the function to the default current schema (and current role, if relevant) are reset (popped).

If SQL authorization mode is not enabled, an attempt to create a function with EXTERNAL SECURITY will result in an error.

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.

Example of declaring a scalar function

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

Example of declaring a table function

```
CREATE FUNCTION PROPERTY_FILE_READER
( FILENAME VARCHAR( 32672 ) )
RETURNS TABLE
  (
     KEY_COL     VARCHAR( 10 ),
     VALUE_COL VARCHAR( 1000 )
  )
LANGUAGE JAVA
PARAMETER STYLE DERBY_JDBC_RESULT_SET
NO SQL
EXTERNAL NAME 'vtis.example.PropertyFileVTI.propertyFileVTI'
```

Example of declaring a function that takes varargs

```
CREATE FUNCTION maximum
( a INT ... )
RETURNS INT
LANGUAGE JAVA
PARAMETER STYLE DERBY
NO SQL
EXTERNAL NAME 'IntFunctions.maximum'
```

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 indexName
ON tableName ( simpleColumnName [ ASC | DESC ]
    [ , simpleColumnName [ 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 does not support indexing on columns with user-defined data types or with the data types LONG VARCHAR, BLOB, CLOB, or XML.

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.

Sorting and ordering of character data is controlled by the collation specified for a database when it is created, as well as the locale of the database. For details, see *collation=collation* attribute and *territory=ll_CC* attribute, as well as the sections "Creating a database with locale-based collation", "Creating a case-insensitive database", and "Character-based collation in Derby" in the *Derby Developer's Guide*.

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. Adding a PRIMARY KEY or UNIQUE constraint when an existing UNIQUE index exists on the same set of columns will result in two physical indexes on the table for the same set of columns. One index is the original UNIQUE index and one is the backing index for the new constraint.

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 creates a Java stored procedure, which you can then call using the CALL PROCEDURE statement.

The procedure owner and the database owner automatically gain the EXECUTE privilege on the procedure, and are able to grant this privilege to other users. The EXECUTE privileges cannot be revoked from the procedure and database owners.

For details on how Derby matches procedures to Java methods, see Argument matching. For information on how stored procedures interact with deferrable constraints, see Deferrable constraints.

Syntax

```
CREATE PROCEDURE procedureName ( [ procedureParameter
   [ , procedureParameter ]* [...] ] )
[ procedureElement ]*
```

An ellipsis (`...`) after the last parameter indicates that the Java method supports trailing optional arguments, called varargs. The ellipsis indicates that the method may be invoked with zero or more trailing values, all having the data type of the last argument.

procedureParameter

```
[ { IN | OUT | INOUT } ] [ parameterName ] dataType
```

The default value for a parameter is IN. A *parameterName* must be unique within a procedure.

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

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

procedureElement

```
{
    [ DYNAMIC ] RESULT SETS integer |
    LANGUAGE JAVA |
    { DETERMINISTIC | NOT DETERMINISTIC } |
    EXTERNAL NAMEsingleQuotedString |
    PARAMETER STYLE { JAVA | DERBY } |
    EXTERNAL SECURITY { DEFINER | INVOKER } |
    { NO SQL | MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA }
}
```

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

DYNAMIC RESULT SETS *integer*

Indicates the estimated upper bound of returned result sets for the procedure. Default is no (zero) dynamic result sets. If the procedure takes varargs, the value must be zero.

LANGUAGE JAVA

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

DETERMINISTIC, NOT DETERMINISTIC

DETERMINISTIC declares that the procedure is deterministic, meaning that with the same set of input values, it always computes the same result. The default is NOT DETERMINISTIC. Derby cannot recognize whether an operation is actually deterministic, so you must take care to specify this element correctly.

EXTERNAL NAME *singleQuotedString*

The *singleQuotedString* specified by the EXTERNAL NAME clause describes the Java method to be called when the procedure is executed.

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.

DERBY
The PARAMETER STYLE must be DERBY if and only if an ellipsis (`...`) appears at the end of the argument list.

EXTERNAL SECURITY

If SQL authorization mode is enabled, a procedure runs by default with the privileges specified for the user who invokes the procedure (invoker's rights). To specify that the procedure should run with the privileges specified for the user who defines the procedure (definer's rights), create the procedure with EXTERNAL SECURITY DEFINER. Those privileges include the right to set the current role to a role for which the definer has privileges. When the procedure is first invoked, no role is set; even if the invoker has set a current role, the procedure running with definer's rights has no current role set initially.

See *derby.database.sqlAuthorization* for details about setting SQL authorization mode.

When a procedure with definer's rights is called, the current default schema is set to the eponymously named schema of the definer. For example, if the defining user is called OWNER, the default schema will also be set to OWNER.

When a procedure with invoker's rights is called, the current default schema and current role are unchanged initially within the procedure. Similarly, if SQL authorization mode is not enabled, the current default schema is unchanged initially within the procedure.

When the call returns, any changes made inside the procedure to the default current schema (and current role, if relevant) are reset (popped).

If SQL authorization mode is not enabled, an attempt to create a procedure with EXTERNAL SECURITY will result in an error.

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

Indicates whether the stored procedure issues any SQL statements and, if so, what type. MODIFIES SQL DATA is the default value. A stored procedure which issues a statement which does not conform to the declared SQL statement level will cause Derby to throw an exception.

NO SQL
Indicates that the stored procedure cannot execute any SQL statements
CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the stored procedure.
READS SQL DATA
Indicates that SQL statements that do not modify SQL data (for example, SELECT statements) can be included in the stored procedure.
MODIFIES SQL DATA
Indicates that the stored procedure can execute any SQL statement.

Examples

```
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.example.sales.calculateRevenueByMonth'
```

```
CREATE PROCEDURE VARARGPROC
( IN a INT, IN b INT, IN c BIGINT ... )
LANGUAGE JAVA
PARAMETER STYLE DERBY
READS SQL DATA
EXTERNAL NAME 'Procs.varargProc'
```

CREATE ROLE statement

The CREATE ROLE statement creates an SQL role. Roles are useful for administering privileges when a database has many users.

Only the database owner can create a role.

For more information on roles, see "Using SQL roles" in the *Derby Security Guide*.

Syntax

```
CREATE ROLE roleName
```

Before you issue a CREATE ROLE statement, verify that the *derby.database.sqlAuthorization* property is set to TRUE. The *derby.database.sqlAuthorization* property enables SQL authorization mode.

You cannot create a role name if there is a user by that name. An attempt to create a role name that conflicts with an existing user name raises the *SQLException* X0Y68.

If user names are not controlled by the database owner (or administrator), it may be a good idea to use a naming convention for roles to reduce the possibility of collision with user names.

Derby tries to avoid name collision between user names and role names, but this is not always possible, because Derby has a pluggable authorization architecture. For example, an externally defined user may exist who has never yet connected to the database, created any schema objects, or been granted any privileges. If Derby knows about a user name, it will forbid creating a role with that name. Correspondingly, a user who has the same name as a role will not be allowed to connect. Derby built-in users are checked for collision when a role is created.

A role name cannot start with the prefix SYS (after case normalization). The purpose of this restriction is to reserve a name space for system-defined roles at a later point. Use of the prefix SYS raises the *SQLException* 4293A.

You cannot create a role with the name `PUBLIC` (after case normalization). `PUBLIC` is a reserved authorization identifier. An attempt to create a role with the name `PUBLIC` raises *SQLException* 4251B.

Example of creating a role

```
CREATE ROLE purchases_reader;
```

Examples of invalid role names

```
CREATE ROLE public;   -- throws SQLException;
CREATE ROLE "PUBLIC"; -- throws SQLException;
CREATE ROLE sysrole;  -- throws SQLException;
```

Example of creating a role using a naming convention

The following example uses the convention of giving every role name the suffix `_role`.

```
CREATE ROLE purchases_reader_role;
```

CREATE SCHEMA statement

The CREATE SCHEMA statement creates a schema, which is a way to logically group objects in a single collection and to provide a unique namespace for objects.

Syntax

```
CREATE SCHEMA
{
```

```
    [ schemaName AUTHORIZATION userName ] |
    [ schemaName ] |
    [ AUTHORIZATION userName ]
}
```

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

A schema name cannot start with the prefix SYS (after case normalization). Use of the prefix SYS raises a *SQLException*.

The CREATE SCHEMA statement is subject to access control when the *derby.database.sqlAuthorization* property is set to `true` for the database or system. Only the database owner can create a schema with a name different from the current user name, and only the the database owner can specify

```
AUTHORIZATION userName
```

with a user name other than the current user name.

Note: Although the SQL standard allows you to specify any *authorizationIdentifier* as an AUTHORIZATION argument, Derby allows you to specify only a user, not a role.

CREATE SCHEMA examples

To create a schema for airline-related tables and give the authorization ID `anita` access to all of the objects that use the schema, use the following syntax:

```
CREATE SCHEMA FLIGHTS AUTHORIZATION anita
```

To create a schema employee-related tables, use the following syntax:

```
CREATE SCHEMA EMP
```

To create a schema that uses the same name as the authorization ID `takumi`, use the following syntax:

```
CREATE SCHEMA AUTHORIZATION takumi
```

To create a table called `availability` in the `EMP` and `FLIGHTS` schemas, use the following syntax:

```
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 SEQUENCE statement

The CREATE SEQUENCE statement creates a sequence generator, which is a mechanism for generating exact numeric values, one at a time.

The owner of the schema where the sequence generator lives automatically gains the USAGE privilege on the sequence generator, and can grant this privilege to other users and roles. Only the database owner and the owner of the sequence generator can grant these USAGE privileges. The USAGE privilege cannot be revoked from the schema owner. See GRANT statement and REVOKE statement for more information.

Syntax

```
CREATE SEQUENCE sequenceName [ sequenceElement ]*
```

The sequence name is composed of an optional *schemaName* and a *SQLIdentifier*. If a *schemaName* is not provided, the current schema is the default schema. If a qualified sequence name is specified, the schema name cannot begin with SYS.

sequenceElement

```
{
  AS dataType
  | START WITH signedInteger
  | INCREMENT BY signedInteger
  | MAXVALUE signedInteger | NO MAXVALUE
  | MINVALUE signedInteger | NO MINVALUE
  | CYCLE | NO CYCLE
}
```

If specified, the *dataType* must be an integer type (SMALLINT, INT, or BIGINT). If not specified, the default data type is INT.

If specified, the INCREMENT value is a non-zero number which fits in a *dataType* value. If not specified, the INCREMENT defaults to 1. INCREMENT is the step by which the sequence generator advances. If INCREMENT is positive, the sequence numbers get larger over time. If INCREMENT is negative, the sequence numbers get smaller.

If specified, MINVALUE must be an integer which fits in a *dataType* value. If MINVALUE is not specified, or if NO MINVALUE is specified, MINVALUE defaults to the smallest negative number which fits in a *dataType* value.

If specified, MAXVALUE may not be greater than the largest positive integer that fits in a *dataType* value. If MAXVALUE is not specified, or if NO MAXVALUE is specified, MAXVALUE defaults to the largest positive integer which fits in a *dataType* value. MAXVALUE must be greater than MINVALUE.

The START WITH clause specifies the initial value of the sequence generator. This value must fall between MINVALUE and MAXVALUE. If the START WITH clause is not specified, the initial value defaults to be:

- MINVALUE if INCREMENT is positive
- MAXVALUE if INCREMENT is negative

The CYCLE clause controls what happens when the sequence generator exhausts its range and wraps around. If CYCLE is specified, the wraparound behavior is to reinitialize the sequence generator to its MINIMUM or MAXIMUM value. If NO CYCLE is specified, Derby throws an exception when the generator wraps around. The default behavior is NO CYCLE. Note that cycling restarts from the minimum or maximum value, not from the start value.

To retrieve the next value from a sequence generator, use a NEXT VALUE FOR expression.

Performance

To boost performance and concurrency, Derby preallocates ranges of upcoming values for sequences. The lengths of these ranges can be configured by adjusting the value of the derby.language.sequence.preallocator property.

Examples

The following statement creates a sequence generator of type INT, with a start value of -2147483648 (the smallest INT value). The value increases by 1, and the last legal value is the largest possible INT. If NEXT VALUE FOR is invoked on the generator again, Derby throws an exception.

```
CREATE SEQUENCE order_id;
```

The following statement creates a sequence of type BIGINT with a start value of 3,000,000,000. The value increases by 1, and the last legal value is the largest possible BIGINT. If NEXT VALUE FOR is invoked on the generator again, Derby throws an exception.

```
CREATE SEQUENCE order_entry_id
AS BIGINT
START WITH 3000000000;
```

CREATE SYNONYM statement

The CREATE SYNONYM statement provides 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 synonymName FOR { viewName | tableName }
```

The *synonymName* in the statement represents the synonym name you are giving the target table or view, while the *viewName* or *tableName* represents the original name of the target table or view.

Example

```
CREATE SYNONYM SAMP.T1 FOR SAMP.TABLEWITHLONGNAME
```

CREATE TABLE statement

The 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).

The table owner and the database owner automatically gain the following privileges on the table and are able to grant these privileges to other users:

- INSERT
- SELECT
- REFERENCES

- TRIGGER
- UPDATE

These privileges cannot be revoked from the table and database owners.

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

There are two different variants of the CREATE TABLE statement, depending on whether you are specifying the column definitions and constraints, or whether you are modeling the columns after the results of a query expression:

```
CREATE TABLE tableName
{
    ( { columnDefinition | tableLevelConstraint }
    [ , { columnDefinition | tableLevelConstraint } ] * )
  |
    [ ( simpleColumnName [ , simpleColumnName ] * ) ]
    AS queryExpression
    WITH NO DATA
}
```

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 and with cycle option.
CREATE TABLE GROUPS
 (GROUP_ID SMALLINT NOT NULL GENERATED ALWAYS AS IDENTITY
 (START WITH 5, INCREMENT BY 5, CYCLE), ADDRESS VARCHAR(100), PHONE
 VARCHAR(15));
```

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

CREATE TABLE ... AS ...

With the alternate form of the CREATE TABLE statement, the column names and/or the column data types can be specified by providing a query. The columns in the query result are used as a model for creating the columns in the new table.

If no column names are specified for the new table, then all the columns in the result of the query expression are used to create same-named columns in the new table, of the corresponding data type(s). If one or more column names are specified for the new table, then the same number of columns must be present in the result of the query expression;

the data types of those columns are used for the corresponding columns of the new table.

The WITH NO DATA clause specifies that the data rows which result from evaluating the query expression are not used; only the names and data types of the columns in the query result are used. The WITH NO DATA clause **must** be specified; in a future release, Derby may be modified to allow the WITH DATA clause to be provided, which would indicate that the results of the query expression should be inserted into the newly-created table. In the current release, however, only the WITH NO DATA form of the statement is accepted.

Example

```
-- create a new table using all the columns and data types
-- from an existing table:
CREATE TABLE T3 AS SELECT * FROM T1 WITH NO DATA;
-- create a new table, providing new names for the columns, but
-- using the data types from the columns of an existing table:
CREATE TABLE T3 (A,B,C,D,E) AS SELECT * FROM T1 WITH NO DATA;
-- create a new table, providing new names for the columns,
-- using the data types from the indicated columns of an existing table:
CREATE TABLE T3 (A,B,C) AS SELECT V,DP,I FROM T1 WITH NO DATA;
-- This example shows that the columns in the result of the
-- query expression may be unnamed expressions, but their data
-- types can still be used to provide the data types for the
-- corresponding named columns in the newly-created table:
CREATE TABLE T3 (X,Y) AS SELECT 2*I,2.0*F FROM T1 WITH NO DATA;
```

columnDefinition:

```
simpleColumnName [ dataType ]
    [ columnLevelConstraint ]*
    [ [ WITH ] DEFAULT defaultConstantExpression
       | generatedColumnSpec
       | generationClause
    ]
    [ columnLevelConstraint ]*
```

The syntax of *dataType* is described in Data types. The *dataType* can be omitted only if you specify a *generationClause*. If you omit the *dataType*, the type of the generated column is the type of the *generationClause*. If you specify both a *dataType* and a *generationClause*, the type of the *generationClause* must be assignable to *dataType*.

The syntaxes of *columnLevelConstraint* and *tableLevelConstraint* are described in CONSTRAINT clause.

Column default

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

```
defaultConstantExpression:
          NULL
        | CURRENT { SCHEMA | SQLID }
        | USER | CURRENT_USER | SESSION_USER | CURRENT_ROLE
        | DATE
        | TIME
        | TIMESTAMP
        | CURRENT DATE | CURRENT_DATE
        | CURRENT TIME | CURRENT_TIME
        | CURRENT TIMESTAMP | CURRENT_TIMESTAMP
        | literal
```

For details about Derby *literal* values, see Data types.

The values in a *defaultConstantExpression* must be compatible in type with the column, but a *defaultConstantExpression* has the following additional type restrictions:

- If you specify USER, CURRENT_USER, SESSION_USER, or CURRENT_ROLE, the column must be a character column whose length is at least 8.
- If you specify CURRENT SCHEMA or CURRENT SQLID, the column must be a character column whose length is at least 128.
- If the column is an integer type, the default value must be an integer literal.
- If the column is a decimal type, the scale and precision of the default value must be within those of the column.

generatedColumnSpec:

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

Identity column attributes

A table can have at most one identity column.

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, the amount of the increment is 1, and does not cycle. You can specify non-default values for the initial value, the interval amount and the cycle option when you define the column with the key words START WITH, INCREMENT BY and CYCLE. And if you specify a negative number for the increment value, Derby*decrements* the value with each insert. If this value is positive, Derby increments the value with each insert. A value of 0 raises a statement exception.

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. The following table shows the supported ranges.

Table 2. 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 function 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.

The CYCLE clause controls what happens when the identity column exhausts its range and wraps around. If CYCLE is specified, the wraparound behavior is to reinitialize the the value of identity column to its minimum or maximum value. If CYCLE is not specified Derby throws an exception when the generator wraps around. In default behavior identity column does not cycle. Note that cycling restarts from the minimum or maximum value, not from the start value.

When an insert happens within a *triggeredSQLStatement*, the value inserted by the *triggeredSQLStatement* 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 *triggeredSQLStatement* 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 1

```
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');
```

Example 2 with cycle option

```
create table greetings
    (i int generated by default as identity (START WITH 2147483647,
 INCREMENT BY 1, CYCLE),
    j int);
-- when values are inserted, the identity column value restarts with
 minimum value:
insert into t( b ) values ( 1 ), ( 2 ), ( 3 );
```

generationClause:

```
GENERATED ALWAYS AS ( valueExpression )
```

A *valueExpression* is an *expression* that resolves to a single value, with some limitations that are described here. See SQL expressions for more information about *expression*s.

References

The *generationClause* may reference other non-generated columns in the table, but it must not reference any generated column. The *generationClause* must not reference a column in another table.

Functions

The *generationClause* may invoke user-coded functions, if the functions meet the following requirements:

- The functions must not read or write SQL data.
- The functions must have been declared DETERMINISTIC.
- The functions must not invoke any of the following possibly non-deterministic system functions:

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- CURRENT_ROLE
- CURRENT SCHEMA
- CURRENT SQLID
- SESSION_USER

Subqueries

The *generationClause* must not include subqueries.

Foreign keys

If the generated column is part of a foreign key that references another table, the referential action must not specify SET NULL or SET DEFAULT, and the update rule must not specify ON UPDATE CASCADE.

Example

```
CREATE TABLE employee
(
  employeeID           int,
  name                 varchar( 50 ),
  caseInsensitiveName  GENERATED ALWAYS AS( UPPER( name ) )
);
CREATE INDEX caseInsensitiveEmployeeName ON employee( caseInsensitiveName
 );
```

CREATE TRIGGER statement

The CREATE TRIGGER statement creates a trigger, which 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 where you are the schema owner. To create a trigger on a table that you do not own, you must be granted the TRIGGER privilege on that table. The database owner can also create triggers on any table in any schema.

A trigger operates with the privileges of the owner of the trigger. See "Configuring fine-grained user authorization" and "Privileges on views, triggers, constraints, and generated columns" in the *Derby Security Guide* for details.

The trigger does not need to reside in the same schema as the table on which the trigger 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 columnName [ , columnName ]* ] }
ON tableName
[ referencingClause ]
```

```
[ FOR EACH { ROW | STATEMENT } ] [ MODE DB2SQL ]
[ WHEN ( booleanExpression )
 ]
triggeredSQLStatement
```

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 *triggeredSQLStatement*s need to refer to data that is currently being changed by the database event that caused them to fire. The *triggeredSQLStatement* 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 *triggeredSQLStatement* 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 *correlationName* .

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 *triggeredSQLStatement*:

```
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 *triggeredSQLStatement* 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 correlationName

For example:

```
REFERENCING OLD_TABLE AS DeletedHotels
```

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

```
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.

The transition tables or transition variables defined in the REFERENCING clause can be referenced from the WHEN clause.

Statement versus row triggers
You have the option to specify whether a trigger is a *statement trigger* or a *row trigger*. If it is not specified in the CREATE TRIGGER statement via FOR EACH clause, then the trigger is a *statement trigger* by default.

- *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.

triggeredSQLStatement
The action defined by the trigger is called the *triggeredSQLStatement* (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 reference a table in the SESSION schema (such as a temporary table; see DECLARE GLOBAL TEMPORARY TABLE statement).
- Before triggers cannot have INSERT, UPDATE or DELETE statements as their action.
- Before triggers cannot call procedures that modify SQL data as their action.
- The NEW variable of a Before trigger cannot reference a generated column.

For more information on *triggeredSQLStatement*s, see "Programming trigger actions" in 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.

Examples

```
-- 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
  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
  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 function
- CURRENT_TIME function
- CURRENT_TIMESTAMP function
- CURRENT_USER function
- SESSION_USER function
- USER function

referencingClause:

```
REFERENCING
{
    { OLD | NEW } [ ROW ] [ AS ] correlationName
        [ { OLD | NEW } [ ROW ] [ AS ] correlationName ]
  |
    { OLD_TABLE | NEW_TABLE } [ AS ] identifier
        [ { OLD_TABLE | NEW_TABLE } [ AS ] identifier ]
}
```

Note: The **OLD_TABLE | NEW_TABLE** syntax is deprecated since it is not SQL compliant and is intended for backward compatibility and DB2 compatibility.

WHEN clause:

The WHEN clause is an optional part of a CREATE TRIGGER statement.

Syntax

```
WHEN ( booleanExpression )
```

If a trigger has been created with a WHEN clause, and the trigger event takes place, the *triggeredSQLStatement* will be executed only if the *booleanExpression* in the WHEN clause evaluates to TRUE. If it evaluates to FALSE or NULL, the *triggeredSQLStatement* will not be executed.

The transition tables or transition variables defined in the REFERENCING clause can be referenced from the WHEN clause.

The restrictions listed for the *triggeredSQLStatement* in the CREATE TRIGGER statement also apply to the WHEN clause.

Note: The use of a WHEN clause in a CREATE TRIGGER statement is valid only after a database has been fully upgraded to Derby Release 10.11 or higher. (See "Upgrading a database" in the *Derby Developer's Guide* for more information.) This clause has no meaning in a database that is at Release 10.10 or lower.

Example

```
CREATE TRIGGER FLIGHTSUPDATE
    AFTER UPDATE ON FLIGHTS
    REFERENCING OLD AS OLD NEW AS NEW
    FOR EACH ROW
    WHEN (OLD.FLIGHT_ID <> NEW.FLIGHT_ID)
    UPDATE FLIGHTAVAILABILITY
        SET FLIGHT_ID = NEW.FLIGHT_ID
        WHERE FLIGHT_ID = OLD.FLIGHT_ID
```

CREATE TYPE statement

The CREATE TYPE statement creates a user-defined type (UDT). A UDT is a serializable Java class whose instances are stored in columns.

The Java class, specified by the EXTERNAL NAME clause, must implement the *java.io.Serializable* interface.

Syntax

```
CREATE TYPE typeNameEXTERNAL NAMEsingleQuotedString
LANGUAGE JAVA
```

The type name is composed of an optional *schemaName* and a *SQLIdentifier*. If a *schemaName* is not provided, the current schema is the default schema. If a qualified type name is specified, the schema name cannot begin with SYS.

If the Java class specified by the EXTERNAL NAME clause does not implement *java.io.Serializable*, or if it is not public and visible on the classpath, Derby raises an exception when preparing statements which refer to the UDT.

A UDT cannot be cast explicitly to any other type, and no other type can be cast to a UDT.

A UDT has no ordering. This means that you cannot compare and sort UDTs. You cannot use them in expressions involving the <, =, >, IN, BETWEEN, and LIKE operators. You cannot use UDTs in aggregates, DISTINCT expressions, and GROUP/ORDER BY clauses. You cannot build indexes on them.

You can use subtypes in UDTs. That is, if you use the CREATE TYPE statement to bind a class named C to a UDT, you can populate that UDT value with an instance of any subclass of C.

Example

```
CREATE TYPE price
EXTERNAL NAME 'com.example.types.Price'
LANGUAGE JAVA
```

Using user-defined types

You can create tables and views with columns that have UDTs. For example:

```
CREATE TABLE order
(
    orderID INT GENERATED ALWAYS AS IDENTITY,
    customerID INT REFERENCES customer( customerID ),
    totalPrice typeSchema.price
);
```

Although UDTs have no natural order, you can use generated columns to provide useful sort orders:

```
ALTER TABLE order
  ADD COLUMN normalizedValue DECIMAL( 31, 5 ) GENERATED ALWAYS AS
    ( convert( 'EUR', TIMESTAMP('2005-01-01 09:00:00'), totalPrice ) );
CREATE INDEX normalizedOrderPrice ON order( normalizedValue );
```

You can use factory functions to construct UDTs. For example:

```
INSERT INTO order( customerID, totalPrice )
  VALUES ( 12345,
           makePrice( 'USD',
                      CAST( 9.99 AS DECIMAL( 31, 5 ) ),
                      TIMESTAMP('2009-10-16 14:24:43') ) );
```

Once a UDT column has been populated, you can use it in other INSERT and UPDATE statements. For example:

```
INSERT INTO backOrder SELECT * from order;

UPDATE order SET totalPrice = ( SELECT todaysDiscount FROM discount );
UPDATE order SET totalPrice = adjustForInflation( totalPrice );
```

Using functions, you can access fields inside UDTs in a SELECT statement:

```
SELECT getCurrencyCode( totalPrice ) from order;
```

You can use JDBC API *setObject()* and *getObject()* methods to store and retrieve values of UDTs. For example:

```
PreparedStatement ps = conn.prepareStatement( "SELECT * from order" );
ResultSet rs = ps.executeQuery();

while( rs.next() )
{
    int    orderID = rs.getInt( 1 );
    int    customerID = rs.getInt( 2 );
    Price  totalPrice = (Price) rs.getObject( 3 );
    ...
}
```

CREATE VIEW statement

The CREATE VIEW statement creates a view, which is a virtual table 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*.

A view operates with the privileges of the owner of the view. See "Configuring fine-grained user authorization" and "Privileges on views, triggers, constraints, and generated columns" in the *Derby Security Guide* for details.

The view owner automatically gains the SELECT privilege on the view. The SELECT privilege cannot be revoked from the view owner. The database owner automatically gains the SELECT privilege on the view and is able to grant this privilege to other users. The SELECT privilege cannot be revoked from the database owner.

The view owner can only grant the SELECT privilege to other users if the view owner also owns the underlying objects.

If the underlying objects that the view references are not owned by the view owner, the view owner must be granted the appropriate privileges. For example, if the authorization ID `user2` attempts to create a view called `user2.v2` that references table `user1.t1` and function `user1.f_abs()`, then `user2` must have the SELECT privilege on table `user1.t1` and the EXECUTE privilege on function `user1.f_abs()`.

The privilege to grant the SELECT privilege cannot be revoked. If a required privilege on one of the underlying objects that the view references is revoked, then the view is dropped.

Syntax

```
CREATE VIEW viewName
    [ ( simpleColumnName [ , simpleColumnName ]* ) ]
AS query [ ORDER BY clause ]
         [ result offset clause ]
         [ fetch first clause ]
```

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.

Examples

```
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.*

DECLARE GLOBAL TEMPORARY TABLE statement

The DECLARE GLOBAL TEMPORARY TABLE statement defines a temporary table for the current connection.

Temporary 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 tempTableName
    { columnDefinition [ , columnDefinition ]* }
[ ON COMMIT { DELETE | PRESERVE } ROWS ]
NOT LOGGED [ ON ROLLBACK DELETE ROWS ]
```

tempTableName

Names the temporary table. If a *schemaName* other than SESSION is specified, an error will occur (SQLSTATE 428EK). If the *schemaName* 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.

columnDefinition

See *columnDefinition* for CREATE TABLE for more information on *columnDefinition*. DECLARE GLOBAL TEMPORARY TABLE does not allow *generatedColumnSpec* in the *columnDefinition*.

Data type

Supported data types are:

- BIGINT
- CHAR
- DATE
- DECIMAL
- DOUBLE
- 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: Temporary tables can be declared only 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

Derby does not support the following features on temporary tables. Some of these features are specific to temporary tables and some are specific to Derby.

Temporary tables cannot be specified in the following statements:

- ALTER TABLE
- CREATE INDEX
- CREATE SYNONYM
- CREATE TRIGGER
- CREATE VIEW

- GRANT
- LOCK TABLE
- RENAME
- REVOKE

You cannot use the following features with temporary tables:

- Synonyms, triggers and views on SESSION schema tables (including physical tables and temporary tables)
- Caching statements that reference SESSION schema tables and views
- Temporary tables cannot be specified in referential constraints and primary keys
- Temporary tables cannot be referenced in a *triggeredSQLStatement* or in a WHEN clause
- Check constraints on columns
- Generated-column-spec
- Importing into temporary tables

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

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

- BLOB
- CHAR FOR BIT DATA
- CLOB
- LONG VARCHAR
- LONG VARCHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- XML

Global temporary tables can be used in XA transactions, but can be declared and accessed only within the scope of a single XA transaction. Derby can support access to the table only until one of the following methods of the *javax.transaction.xa.XAResource* interface is called:

- *XAResource.end*
- *XAresource.prepare*
- *XAResource.commit*

When the XA transaction commits or aborts, the temporary table is dropped.

DELETE statement

The DELETE statement removes rows from a table.

Syntax

```
{
    DELETE FROM tableName [ [ AS ] correlationName ]
        [ WHERE clause ]
  |
    DELETE FROM tableNameWHERE CURRENT OF clause
}
```

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

The second syntactical form, called a positioned delete, deletes the current row of an open, updatable cursor. 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());
```

Statement dependency system

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

The positioned delete statement depends on the cursor and any tables the cursor references. You can compile a positioned delete even if the cursor 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.

DROP statements

Use DROP statements to remove functions, indexes, procedures, roles, schemas, synonyms, tables, triggers, and views.

DROP DERBY AGGREGATE statement

The DROP DERBY AGGREGATE statement removes the specified user-defined aggregate (UDA).

A UDA is created by a CREATE DERBY AGGREGATE statement.

Syntax

```
DROP DERBY AGGREGATE aggregateName RESTRICT
```

The RESTRICT keyword is required. CASCADE semantics are not supported. That is, Derby will not track down and drop orphaned objects.

Dropping a UDA implicitly drops all USAGE privileges that reference it. See GRANT statement and REVOKE statement for more information.

Derby raises an error if a trigger or view references the UDA.

Example

```
DROP DERBY AGGREGATE mode RESTRICT;
```

DROP FUNCTION statement

The DROP FUNCTION statement removes the specified Java function.

A function is created by a CREATE FUNCTION statement.

Syntax

```
DROP FUNCTION functionName
```

The argument identifies the particular function to be dropped and is valid only if there is exactly one function instance with the *functionName* in the schema. The identified function can have any number of parameters defined for it.

An error will occur in any of the following circumstances:

- If no function with the indicated name exists in the named or implied schema (the error is SQLSTATE 42704)

- If there is more than one specific instance of the function in the named or implied schema
- If you try to drop a user-defined function that is invoked in the *generationClause* of a generated column
- If you try to drop a user-defined function that is invoked in a view or trigger

DROP INDEX statement

The DROP INDEX statement removes the specified index.

An index is created by a CREATE INDEX statement.

Syntax

```
DROP INDEX indexName
```

Examples

```
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

The DROP PROCEDURE statement removes the specified Java stored procedure.

A stored procedure is created by a CREATE PROCEDURE statement and is called by a CALL (PROCEDURE) statement.

Syntax

```
DROP PROCEDURE procedureName
```

Identifies the particular procedure to be dropped, and is valid only if there is exactly one procedure instance with the *procedureName* in the schema. The identified procedure can have any number of parameters defined for it.

An error will occur in any of the following circumstances:

- If no procedure with the indicated name exists in the named or implied schema (the error is SQLSTATE 42704)
- If there is more than one specific instance of the procedure in the named or implied schema
- If you try to drop a user-defined procedure that is invoked in a trigger

DROP ROLE statement

The DROP ROLE statement removes the specified SQL role.

A role is created by a CREATE ROLE statement.

Only the database owner can drop a role.

For more information on roles, see "Using SQL roles" in the *Derby Security Guide*.

Syntax

```
DROP ROLE roleName
```

Dropping a role has the effect of removing the role from the database dictionary. This means that no session user can henceforth set that role (see SET ROLE statement), and any existing sessions that have that role as the current role (see CURRENT_ROLE

function) will now have a NULL CURRENT_ROLE. Dropping a role also has the effect of revoking that role from any user and role it has been granted to. See REVOKE statement for information on how revoking a role may impact any dependent objects.

Example

```
DROP ROLE reader;
```

DROP SCHEMA statement

The DROP SCHEMA statement removes the specified schema.

A schema is created by a CREATE SCHEMA statement.

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.

Example

```
-- 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 SEQUENCE statement

The DROP SEQUENCE statement removes the specified sequence generator.

A sequence generator is created by a CREATE SEQUENCE statement.

Syntax

```
DROP SEQUENCE sequenceName RESTRICT
```

The RESTRICT keyword is required. If a trigger or view references the sequence generator, Derby throws an exception.

Dropping a sequence generator implicitly drops all USAGE privileges that reference it.

Example

```
DROP SEQUENCE order_id RESTRICT;
```

DROP SYNONYM statement

The DROP SYNONYM statement removes the specified synonym from a table or view.

A synonym is created by a CREATE SYNONYM statement.

An error will occur if there are any views or triggers dependent on the synonym.

Syntax

```
DROP SYNONYM synonymName
```

DROP TABLE statement

The DROP TABLE statement removes the specified table.

A table is created by a CREATE TABLE statement.

Syntax

```
DROP TABLE tableName
```

Statement dependency system

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

The DROP TABLE statement will also generate an error if the table is used in a view, or if a trigger defined on another table references the table in its trigger action.

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

The DROP TRIGGER statement removes the specified trigger.

A trigger is created by a CREATE TRIGGER statement.

Syntax

```
DROP TRIGGER triggerName
```

Example

```
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 TYPE statement

The DROP TYPE statement removes the specified user-defined type (UDT).

A UDT is created by a CREATE TYPE statement.

Syntax

```
DROP TYPE typeName RESTRICT
```

The RESTRICT keyword is required. CASCADE semantics are not supported. That is, Derby will not track down and drop orphaned objects.

Dropping a UDT implicitly drops all USAGE privileges that reference it.

You cannot drop a type if it would make another SQL object unusable. This includes the following restrictions:

- Table columns: No table columns have this UDT.
- Views: No view definition involves expressions which have this UDT.
- Triggers: No trigger definition involves expressions which have this UDT.
- Constraints: No constraints reference expressions of this UDT.
- Generated columns: No generated columns reference expressions of this UDT.
- Routines: No functions or procedures have arguments or return values of this UDT.
- Table Functions: No table functions return tables with columns of this UDT.

Example

```
DROP TYPE price RESTRICT;
```

DROP VIEW statement

The DROP VIEW statement removes the specified view.

A view is created by a CREATE VIEW statement.

Syntax

```
DROP VIEW viewName
```

Example

```
DROP VIEW AnIdentifier
```

Statement dependency system

Any statements referencing the view are invalidated on a DROP VIEW statement. DROP VIEW fails if there are any views, triggers, or open cursors dependent on the view.

Normally, you must drop a view before you drop any objects that the view depends on. However, if you issue an ALTER TABLE DROP COLUMN command with the CASCADE option, any views that depend on the column will be dropped. Also, if you use a REVOKE statement to revoke privileges on objects that a view depends on, the view will be dropped. Similarly, if you use a DROP ROLE statement to drop a role that has privileges on objects that a view depends on, the view will be dropped.

GRANT statement

The GRANT statement gives privileges to a specific user or role, or to all users, to perform actions on database objects.

You can also use the GRANT statement to grant a role to a user, to PUBLIC, or to another role.

The following types of privileges can be granted:

- Delete data from a specific table.
- Insert data into a specific table.
- Create a foreign key reference to the named table or to a subset of columns from a table.
- Select data from a table, view, or a subset of columns in a table.
- Create a trigger on a table.
- Update data in a table or in a subset of columns in a table.
- Run a specified function or procedure.
- Use a sequence generator, a user-defined type, or a user-defined aggregate.

Before you issue a GRANT statement, check that the *derby.database.sqlAuthorization* property is set to `true`. The *derby.database.sqlAuthorization* property enables the SQL Authorization mode.

You can grant privileges on an object if you are the owner of the object or the database owner. See the CREATE statement for the database object that you want to grant privileges on for more information.

The syntax that you use for the GRANT statement depends on whether you are granting privileges to a schema object or granting a role.

For more information on using the GRANT statement, see "Using fine-grained user authorization" in the *Derby Security Guide*.

Syntax for tables

```
GRANT privilegeType ON [ TABLE ] { tableName | viewName } TO grantees
```

Syntax for routines

```
GRANT EXECUTE ON FUNCTION functionName TO grantees
```

```
GRANT EXECUTE ON PROCEDURE procedureName TO grantees
```

Syntax for sequence generators

```
GRANT USAGE ON SEQUENCE sequenceName TO grantees
```

In order to use a sequence generator, you must have the USAGE privilege on it. This privilege can be granted to users and to roles. See CREATE SEQUENCE statement for more information.

Syntax for user-defined types

```
GRANT USAGE ON TYPE typeName TO grantees
```

In order to use a user-defined type, you must have the USAGE privilege on it. This privilege can be granted to users and to roles. See CREATE TYPE statement for more information.

Syntax for user-defined aggregates

```
GRANT USAGE ON DERBY AGGREGATE aggregateName TO grantees
```

In order to use a user-defined aggregate, you must have the USAGE privilege on it. This privilege can be granted to users and to roles. See CREATE DERBY AGGREGATE statement for more information.

Syntax for roles

```
GRANT roleName [ , roleName ]* TO grantees
```

Before you can grant a role to a user or to another role, you must create the role using the CREATE ROLE statement. Only the database owner can grant a role.

A role A *contains* another role B if role B is granted to role A, or is contained in a role C granted to role A. Privileges granted to a contained role are inherited by the containing roles. So the set of privileges identified by role A is the union of the privileges granted to role A and the privileges granted to any contained roles of role A.

privilegeType

```
{ ALL PRIVILEGES | privilegeList }
```

privilegeList

```
tablePrivilege [ , tablePrivilege ]*
```

tablePrivilege

```
  DELETE |
  INSERT |
  REFERENCES [ columnList ] |
  SELECT [ columnList ] |
  TRIGGER |
  UPDATE [ columnList ]
```

columnList

```
  ( columnIdentifier [ , columnIdentifier ]* )
```

Use the ALL PRIVILEGES privilege type to grant all of the privileges to the user or role for the specified table. You can also grant one or more table privileges by specifying a *privilegeList*.

Use the DELETE privilege type to grant permission to delete rows from the specified table.

Use the INSERT privilege type to grant permission to insert rows into the specified table.

Use the REFERENCES privilege type to grant permission to create a foreign key reference to the specified table. If a *columnList* is specified with the REFERENCES privilege, the permission is valid on only the foreign key reference to the specified columns.

Use the SELECT privilege type to grant permission to perform SELECT statements or *selectExpressions* on a table or view. If a column list is specified with the SELECT privilege, the permission is valid on only those columns. If no column list is specified, then the privilege is valid on all of the columns in the table.

For queries that do not select a specific column from the tables involved in a SELECT statement or *selectExpression* (for example, queries that use `COUNT(*)`), the user must have at least one column-level SELECT privilege or table-level SELECT privilege.

Use the TRIGGER privilege type to grant permission to create a trigger on the specified table.

Use the UPDATE privilege type to grant permission to use the UPDATE statement on the specified table. If a column list is specified, the permission applies only to the specified columns. To update a row using a statement that includes a WHERE clause, you must have the SELECT privilege on the columns in the row that you want to update.

grantees

```
{ authorizationIdentifier | roleName | PUBLIC }
[ , { authorizationIdentifier | roleName | PUBLIC } ]*
```

You can grant privileges or roles to specific users or roles or to all users. Use the keyword PUBLIC to specify all users. When PUBLIC is specified, the privileges or roles affect all current and future users. The privileges granted to PUBLIC and to individual users or roles are independent privileges. For example, a SELECT privilege on table `t` is granted to both PUBLIC and to the authorization ID `harry`. The SELECT privilege is later revoked from the authorization ID `harry`, but Harry can access the table `t` through the PUBLIC privilege.

Either the object owner or the database owner can grant privileges to a user or to a role. Only the database owner can grant a role to a user or to another role.

Examples

To grant the SELECT privilege on table `t` to the authorization IDs `maria` and `harry`, use the following syntax:

```
GRANT SELECT ON TABLE t TO maria,harry
```

To grant the UPDATE and TRIGGER privileges on table `t` to the authorization IDs `anita` and `zhi`, use the following syntax:

```
GRANT UPDATE, TRIGGER ON TABLE t TO anita,zhi
```

To grant the SELECT privilege on table `s.v` to all users, use the following syntax:

```
GRANT SELECT ON TABLE s.v to PUBLIC
```

To grant the EXECUTE privilege on procedure `p` to the authorization ID `george`, use the following syntax:

```
GRANT EXECUTE ON PROCEDURE p TO george
```

To grant the role `purchases_reader_role` to the authorization IDs `george` and `maria`, use the following syntax:

```
GRANT purchases_reader_role TO george,maria
```

To grant the SELECT privilege on table `t` to the role `purchases_reader_role`, use the following syntax:

```
GRANT SELECT ON TABLE t TO purchases_reader_role
```

To grant the USAGE privilege on the sequence generator `order_id` to the role `sales_role`, use the following syntax:

```
GRANT USAGE ON SEQUENCE order_id TO sales_role;
```

To grant the USAGE privilege on the user-defined type `price` to the role `finance_role`, use the following syntax:

```
GRANT USAGE ON TYPE price TO finance_role;
```

To grant the USAGE privilege on the user-defined aggregate `types.maxPrice` to the role `sales_role`, use the following syntax:

```
GRANT USAGE ON DERBY AGGREGATE types.maxPrice TO sales_role;
```

INSERT statement

The INSERT statement creates one or more 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.

Whenever you insert into a table which has generated columns, Derby calculates the values of those columns.

Syntax

```
INSERT INTO tableName
    [ ( simpleColumnName [ , simpleColumnName ]* ) ]
          query [ ORDER BY clause ]
                [ result offset clause ]
                [ fetch first clause ]
```

The *query* can be:

- A *selectExpression*
- A single-row or multiple-row VALUES expression

  Single-row and multiple-row VALUES expressions 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.

  The DEFAULT literal is the only value which you can directly insert into a generated column.
- UNION expressions

When you want insertion to happen with a specific ordering (for example, in conjunction with auto-generated keys), it can be useful to specify an ORDER BY clause on the result set to be inserted.

If the *query* is a VALUES expression, it cannot contain or be followed by an ORDER BY, result offset, or fetch first clause. However, if the VALUES expression does not contain

the DEFAULT keyword, the VALUES clause can be put in a subquery and ordered, as in the following statement:

```
INSERT INTO t SELECT * FROM (VALUES 'a','c','b') t ORDER BY 1;
```

Examples

```
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)

-- Create an AIRPORTS table and insert into it
-- some of the fields from the CITIES table, with the airport
-- codes sorted alphabetically
CREATE TABLE AIRPORTS (
  AIRPORT_ID INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY
    PRIMARY KEY,
  AIRPORT VARCHAR(3),
  CITY VARCHAR(24) NOT NULL,
  COUNTRY VARCHAR(26) NOT NULL
);

INSERT INTO AIRPORTS (AIRPORT, CITY, COUNTRY)
  SELECT AIRPORT, CITY_NAME, COUNTRY FROM CITIES
  ORDER BY AIRPORT;
```

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 statement. 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.

LOCK TABLE statement

The LOCK TABLE statement explicitly acquires a shared or exclusive table lock on the specified table.

The table lock lasts until the end of the current transaction.

To lock a table, you must be either the database owner or the table owner.

Explicitly locking a table is useful to:

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

You cannot lock system tables with this statement.

Syntax

```
LOCK TABLE tableName IN { SHARE | EXCLUSIVE } MODE
```

After a table is locked in either mode, a transaction does not acquire any subsequent row-level locks on a table. For example, if a transaction locks the entire `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 the `Flights` table forces the exclusive lock to be table-level as well.

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

Examples

To lock the entire `Flights` table in share mode to avoid a large number of row locks, use the following statement:

```
LOCK TABLE Flights IN SHARE MODE;
SELECT *
FROM Flights
WHERE orig_airport > 'OOO';
```

You have a transaction with multiple UPDATE statements. Since each of the individual statements acquires only a few row-level locks, the transaction will not automatically upgrade the locks to a table-level lock. However, collectively the UPDATE statements acquire and release a large number of locks, which might result in deadlocks. For this type of transaction, you can acquire an exclusive table-level lock at the beginning of the transaction. For example:

```
LOCK TABLE FlightAvailability IN EXCLUSIVE MODE;
UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-03-31');

UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-04-11');

UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-04-12');

UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-04-15');
```

If a transaction needs to look at a table before updating the table, acquire an exclusive lock before selecting to avoid deadlocks. For example:

```
LOCK TABLE Maps IN EXCLUSIVE MODE;
```

```
SELECT MAX(map_id) + 1 FROM Maps;
-- INSERT INTO Maps . . .
```

MERGE statement

The MERGE statement scans a table and either INSERTs, UPDATEs, or DELETEs rows depending on whether the rows satisfy a specified condition.

Syntax

```
MERGE INTO targetTable [ [ AS ] targetCorrelationName ]
USING sourceTable [ [ AS ] sourceCorrelationName ]
ON searchCondition mergeWhenClause [ mergeWhenClause ]*
```

Both *targetTable* and *sourceTable* are *tableName*s.

targetTable must identify a base table. *targetTable* may not be a transition table in a triggered statement, and it may not be a synonym.

sourceTable must identify a base table or a table function, and it may not be a synonym.

Both *targetCorrelationName* and *sourceCorrelationName* are *correlationName*s.

The unqualified source table name (or its correlation name) may not be the same as the unqualified target table name (or its correlation name).

The *searchCondition* is a *Boolean expression*. Columns referenced by the *searchCondition* must be in either *targetTable* or *sourceTable*. Functions mentioned in the *searchCondition* may not modify SQL data.

The row count for a successful MERGE statement is the total number of rows inserted, updated, and deleted by the statement.

Note: The MERGE statement is valid only after a database has been fully upgraded to Derby Release 10.11 or higher. (See "Upgrading a database" in the *Derby Developer's Guide* for more information.) This statement has no meaning in a database that is at Release 10.10 or lower.

mergeWhenClause

```
mergeWhenMatched | mergeWhenNotMatched
```

mergeWhenMatched

```
WHEN MATCHED [ AND matchRefinement ] THEN { mergeUpdate | DELETE }
```

The *matchRefinement* is a *Boolean expression*. Columns referenced by the *matchRefinement* must be in either *targetTable* or *sourceTable*. Functions mentioned in the *matchRefinement* may not modify SQL data.

mergeWhenNotMatched

```
WHEN NOT MATCHED [ AND matchRefinement ] THEN mergeInsert
```

The *matchRefinement* is a *Boolean expression*. Columns referenced by the *matchRefinement* must be in either *targetTable* or *sourceTable*. Functions mentioned in the *matchRefinement* may not modify SQL data.

Although permitted to do so by the SQL Standard, Derby does not currently support subqueries in WHEN [NOT] MATCHED clauses.

mergeUpdate

```
UPDATE SET column-Name = value [, column-Name = value ]*
```

Columns updated must be columns in *targetTable*.

Functions mentioned in the UPDATE values may not modify SQL data.

On the right side of SET operators for UPDATE actions, DEFAULT is the only value allowed for generated and identity columns.

No list of updated columns may mention the same column more than once.

The data types of updated values must be assignable to the corresponding columns according to the rules documented in Data type assignments and comparison, sorting, and ordering.

mergeInsert

```
INSERT [ ( Simple-column-Name [ , Simple-column-Name ]*  ) ] VALUES (
 value [, value ]* )
```

Columns inserted must be columns in *targetTable*.

Functions mentioned in the INSERT values may not modify SQL data.

No list of inserted columns may mention identity columns, or may mention the same column more than once.

In a VALUES clause, DEFAULT is the only allowed value for generated columns.

The data types of inserted values must be assignable to the corresponding columns according to the rules documented in Data type assignments and comparison, sorting, and ordering.

Required privileges

The user who executes a MERGE statement must have the following privileges. See GRANT statement for information on privileges.

- UPDATE privilege on every updated column of *targetTable*. A blanket UPDATE privilege on the entire *targetTable* would cover this.
- INSERT privilege on *targetTable* if there are WHEN NOT MATCHED clauses.
- DELETE privilege on *targetTable* if there are WHEN MATCHED ... THEN DELETE clauses.
- EXECUTE privilege on all functions mentioned in the Boolean expressions and in the INSERT/UPDATE values.
- USAGE privilege on all sequences and user-defined types mentioned in the Boolean expressions and in the INSERT/UPDATE values. See CREATE SEQUENCE statement and CREATE TYPE statement for more information.
- SELECT privilege on all columns mentioned in the Boolean expressions and the *value* expressions of SET clauses.

MERGE statement behavior

The MERGE statement behaves as described in the following table.

Table 3. Merge statement behavior

Situation or Behavior	Description
Source table is empty	If the *sourceTable* is empty, a "no data" warning is raised with SQLState 02000.
An initial join is performed	Before any changes are made to *targetTable*, the *sourceTable* is joined to the *targetTable* by means of the ON clause. Call this join result J. Let N denote the rows in *sourceTable* missing from this join.

Situation or Behavior	Description
Clause order is important	The *mergeWhenMatched* and *mergeWhenNotMatched* clauses are applied in declaration order.
The first matched clause wins	For each row in J, Derby applies only the first *mergeWhenMatched* clause whose *matchRefinement* is satisfied.
The first not matched clause wins	For each row in N, Derby applies only the first *mergeWhenNotMatched* clause whose *matchRefinement* is satisfied.
Double dipping is not permitted	A cardinality violation is raised if a MERGE statement attempts to change (update or delete) the same row twice. This condition can occur if more than one source row joins to the same target row.

Examples

```
MERGE INTO hotIssues h
USING issues i
ON h.issueID = i.issueID
WHEN MATCHED AND i.lastUpdated = CURRENT_DATE
    THEN UPDATE SET h.lastUpdated = i.lastUpdated
WHEN MATCHED AND i.lastUpdated < CURRENT_DATE THEN DELETE
WHEN NOT MATCHED AND i.lastUpdated = CURRENT_DATE
    THEN INSERT VALUES ( i.issueID, i.lastUpdated );

MERGE INTO companies c
USING adhocInvoices a
ON a.companyName = c.companyName
WHEN NOT MATCHED THEN INSERT ( companyName ) VALUES ( a.companyName );

MERGE INTO warehouse.productList w
USING production.productList p
ON w.productID = p.productID
WHEN MATCHED and w.lastUpdated != p.lastUpdated
    THEN UPDATE SET lastUpdated = p.lastUpdated,
                    description = p.description,
                    price = p.price
WHEN NOT MATCHED
    THEN INSERT values ( p.productID, p.lastUpdated, p.description,
                         p.price );
```

RENAME statements

Use the RENAME statements with indexes, columns, and tables.

RENAME COLUMN statement

The RENAME COLUMN statement renames an existing column in an existing table in any schema (except the schema *SYS*).

To rename a column, you must be either the database owner or the table owner.

Other types of table alterations are possible; see ALTER TABLE statement for more information.

Syntax

```
RENAME COLUMN tableName.simpleColumnName TO simpleColumnName
```

Examples

To rename the *manager* column in table *employee* to *supervisor*, use the following syntax:

```
RENAME COLUMN EMPLOYEE.MANAGER TO SUPERVISOR
```

You can combine ALTER TABLE and RENAME COLUMN to modify a column's data type. To change column *c1* of table *t* to the new data type *NEWTYPE*:

```
ALTER TABLE t ADD COLUMN c1_newtype NEWTYPE
UPDATE t SET c1_newtype = c1
ALTER TABLE t DROP COLUMN c1
RENAME COLUMN t.c1_newtype TO c1
```

Usage notes

Restriction: If a view, trigger, check constraint, or *generationClause* of a generated column references the column, an attempt to rename it will generate an error.
Restriction: The RENAME COLUMN statement is not allowed if there are any open cursors that reference the column that is being altered.
Note: If there is an index defined on the column, the column can still be renamed; the index is automatically updated to refer to the column by its new name

RENAME INDEX statement

The RENAME INDEX statement renames an index in the current schema, which can be any schema except the schema *SYS*.

Syntax

```
RENAME INDEX indexName TO newIndexName
```

Example

```
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

The RENAME TABLE statement renames an existing table in any schema (except the schema *SYS*).

To rename a table, you must be either the database owner or the table owner.

Syntax

```
RENAME TABLE tableName TO newTableName
```

If there is a view 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.

Example

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

See ALTER TABLE statement for more information.

Statement dependency system

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

REVOKE statement

The REVOKE statement removes privileges from a specific user or role, or from all users, to perform actions on database objects.

You can also use the REVOKE statement to revoke a role from a user, from PUBLIC, or from another role.

The following types of privileges can be revoked:

- Delete data from a specific table.
- Insert data into a specific table.
- Create a foreign key reference to the named table or to a subset of columns from a table.
- Select data from a table, view, or a subset of columns in a table.
- Create a trigger on a table.
- Update data in a table or in a subset of columns in a table.
- Run a specified routine (function or procedure).
- Use a sequence generator, a user-defined type, or a user-defined aggregate.

The *derby.database.sqlAuthorization* property must be set to `true` before you can use the GRANT statement or the REVOKE statement. The *derby.database.sqlAuthorization* property enables SQL Authorization mode.

You can revoke privileges for an object if you are the owner of the object or the database owner.

The syntax that you use for the REVOKE statement depends on whether you are revoking privileges to a schema object or revoking a role.

For more information on using the REVOKE statement, see "Using fine-grained user authorization" in the *Derby Security Guide*.

Syntax for tables

```
REVOKE privilegeType ON [ TABLE ] { tableName | viewName } FROM revokees
```

Revoking a privilege without specifying a column list revokes the privilege for all of the columns in the table.

Syntax for routines

```
REVOKE EXECUTE ON FUNCTION functionName FROM revokees RESTRICT
```

```
REVOKE EXECUTE ON PROCEDURE procedureName FROM revokees RESTRICT
```

You must use the RESTRICT clause on REVOKE statements for routines. The RESTRICT clause specifies that the EXECUTE privilege cannot be revoked if the specified routine is used in a view, trigger, or constraint, and the privilege is being revoked from the owner of the view, trigger, or constraint.

Syntax for sequence generators

```
REVOKE USAGE ON SEQUENCE sequenceName FROM revokees RESTRICT
```

In order to use a sequence generator, you must have the USAGE privilege on it. This privilege can be revoked from users and roles. Only RESTRICTed revokes are allowed. This means that the REVOKE statement cannot make a view, trigger, or constraint unusable by its owner. The USAGE privilege cannot be revoked from the schema owner. See CREATE SEQUENCE statement for more information.

Syntax for user-defined types

```
REVOKE USAGE ON TYPE typeName FROM revokees RESTRICT
```

In order to use a user-defined type, you must have the USAGE privilege on it. This privilege can be revoked from users and roles. Only RESTRICTed revokes are allowed. This means that the REVOKE statement cannot make a view, trigger, or constraint unusable by its owner. The USAGE privilege cannot be revoked from the schema owner. See CREATE TYPE statement for more information.

Syntax for user-defined aggregates

```
REVOKE USAGE ON DERBY AGGREGATE aggregateName FROM revokees RESTRICT
```

In order to use a user-defined aggregate, you must have the USAGE privilege on it. This privilege can be revoked from users and roles. Only RESTRICTed revokes are allowed. This means that the REVOKE statement cannot make a view or trigger unusable by its owner. The USAGE privilege cannot be revoked from the schema owner. See CREATE DERBY AGGREGATE statement for more information.

Syntax for roles

```
REVOKE roleName [ , roleName ]* FROM revokees
```

Only the database owner can revoke a role.

privilegeType

```
  ALL PRIVILEGES | privilegeList
```

privilegeList

```
tablePrivilege [ , tablePrivilege ]*
```

tablePrivilege

```
  DELETE |
  INSERT |
  REFERENCES [ columnList ] |
  SELECT [ columnList ] |
  TRIGGER |
  UPDATE [ columnList ]
```

columnList

```
  ( columnIdentifier [ , columnIdentifier ]* )
```

Use the ALL PRIVILEGES privilege type to revoke all of the privileges from the user or role for the specified table. You can also revoke one or more table privileges by specifying a *privilegeList*.

Use the DELETE privilege type to revoke permission to delete rows from the specified table.

Use the INSERT privilege type to revoke permission to insert rows into the specified table.

Use the REFERENCES privilege type to revoke permission to create a foreign key reference to the specified table. If a column list is specified with the REFERENCES privilege, the permission is revoked on only the foreign key reference to the specified columns.

Use the SELECT privilege type to revoke permission to perform SELECT statements on a table or view. If a column list is specified with the SELECT privilege, the permission is revoked on only those columns. If no column list is specified, then the privilege is valid on all of the columns in the table.

Use the TRIGGER privilege type to revoke permission to create a trigger on the specified table.

Use the UPDATE privilege type to revoke permission to use the UPDATE statement on the specified table. If a column list is specified, the privilege is revoked only on the specified columns.

revokees

```
{ authorizationIdentifier | roleName | PUBLIC }
[ , { authorizationIdentifier | roleName | PUBLIC } ]*
```

You can revoke the privileges from specific users or roles or from all users. Use the keyword PUBLIC to specify all users. The privileges revoked from PUBLIC and from individual users or roles are independent privileges. For example, a SELECT privilege on table `t` is revokeed to both PUBLIC and to the authorization ID `harry`. The SELECT privilege is later revoked from the authorization ID `harry`, but the authorization ID `harry` can access the table `t` through the PUBLIC privilege.

You can revoke a role from a role, from a user, or from PUBLIC.

Restriction: You cannot revoke the privileges of the owner of an object.

Prepared statements and open result sets/cursors

Checking for privileges happens at statement execution time, so prepared statements are still usable after a revoke action. If sufficient privileges are still available for the session, prepared statements will be executed, and for queries, a result set will be returned.

Once a result set has been returned to the application (by executing a prepared statement or by direct execution), it will remain accessible even if privileges or roles are revoked in a way that would cause another execution of the same statement to fail.

Cascading object dependencies

For views, triggers, and constraints, if the privilege on which the object depends on is revoked, the object is automatically dropped. Derby does not try to determine if you have other privileges that can replace the privileges that are being revoked. For more information, see "Configuring fine-grained user authorization" and "Privileges on views, triggers, constraints, and generated columns" in the *Derby Security Guide*.

Limitations

The following limitations apply to the REVOKE statement:

Table-level privileges

All of the table-level privilege types for a specified revokeee and table ID are stored in one row in the SYSTABLEPERMS system table. For example, when `user2` is revokeed the SELECT and DELETE privileges on table `user1.t1`, a row is added to the SYSTABLEPERMS table. The GRANTEE field contains `user2` and the TABLEID contains `user1.t1`. The SELECTPRIV and DELETEPRIV fields are set to Y. The remaining privilege type fields are set to N.

When a revokeee creates an object that relies on one of the privilege types, the Derby engine tracks the dependency of the object on the specific row in the SYSTABLEPERMS table. For example, `user2` creates the view `v1` by using the statement `SELECT * FROM user1.t1`, the dependency manager tracks the dependency of view `v1` on the row in SYSTABLEPERMS for GRANTEE(`user2`), TABLEID(`user1.t1`). The dependency manager knows only that the view is dependent on a privilege type in that specific row, but does not track exactly which privilege type the view is dependent on.

When a REVOKE statement for a table-level privilege is issued for a revokeee and table ID, all of the objects that are dependent on the revokeee and table ID are dropped. For example, if `user1` revokes the DELETE privilege on table `t1` from `user2`, the row in SYSTABLEPERMS for GRANTEE(`user2`), TABLEID(`user1.t1`)

is modified by the REVOKE statement. The dependency manager sends a revoke invalidation message to the view `user2.v1` and the view is dropped even though the view is not dependent on the DELETE privilege for GRANTEE(`user2`), TABLEID(`user1.t1`).

Column-level privileges

Only one type of privilege for a specified revokeee and table ID are stored in one row in the SYSCOLPERMS system table. For example, when `user2` is revokeed the SELECT privilege on table `user1.t1` for columns c12 and c13, a row is added to the SYSCOLPERMS. The GRANTEE field contains `user2`, the TABLEID contains `user1.t1`, the TYPE field contains `S`, and the COLUMNS field contains `c12, c13`.

When a revokeee creates an object that relies on the privilege type and the subset of columns in a table ID, the Derby engine tracks the dependency of the object on the specific row in the SYSCOLPERMS table. For example, `user2` creates the view `v1` by using the statement `SELECT c11 FROM user1.t1`, the dependency manager tracks the dependency of view `v1` on the row in SYSCOLPERMS for GRANTEE(`user2`), TABLEID(`user1.t1`), TYPE(S). The dependency manager knows that the view is dependent on the SELECT privilege type, but does not track exactly which columns the view is dependent on.

When a REVOKE statement for a column-level privilege is issued for a revokeee, table ID, and type, all of the objects that are dependent on the revokeee, table ID, and type are dropped. For example, if `user1` revokes the SELECT privilege on column `c12` on table `user1.t1` from `user2`, the row in SYSCOLPERMS for GRANTEE(`user2`), TABLEID(`user1.t1`), TYPE(S) is modified by the REVOKE statement. The dependency manager sends a revoke invalidation message to the view `user2.v1` and the view is dropped even though the view is not dependent on the column `c12` for GRANTEE(`user2`), TABLEID(`user1.t1`), TYPE(S).

Roles

Derby tracks any dependencies on the definer's current role for views, constraints, and triggers. If privileges were obtainable only via the current role when the object in question was defined, that object depends on the current role. The object will be dropped if the role is revoked from the defining user or from PUBLIC, as the case may be. Also, if a contained role of the current role in such cases is revoked, dependent objects will be dropped. Note that dropping may be too pessimistic. This is because Derby does not currently make an attempt to recheck if the necessary privileges are still available in such cases.

Revoke examples

To revoke the SELECT privilege on table `t` from the authorization IDs `maria` and `harry`, use the following syntax:

```
REVOKE SELECT ON TABLE t FROM maria,harry
```

To revoke the UPDATE and TRIGGER privileges on table `t` from the authorization IDs `anita` and `zhi`, use the following syntax:

```
REVOKE UPDATE, TRIGGER ON TABLE t FROM anita,zhi
```

To revoke the SELECT privilege on table `s.v` from all users, use the following syntax:

```
REVOKE SELECT ON TABLE s.v FROM PUBLIC
```

To revoke the UPDATE privilege on columns `c1` and `c2` of table `s.v` from all users, use the following syntax:

```
REVOKE UPDATE (c1,c2) ON TABLE s.v FROM PUBLIC
```

To revoke the EXECUTE privilege on procedure p from the authorization ID `george`, use the following syntax:

```
REVOKE EXECUTE ON PROCEDURE p FROM george RESTRICT
```

To revoke the role `purchases_reader_role` from the authorization IDs `george` and `maria`, use the following syntax:

```
REVOKE purchases_reader_role FROM george,maria
```

To revoke the SELECT privilege on table `t` from the role `purchases_reader_role`, use the following syntax:

```
REVOKE SELECT ON TABLE t FROM purchases_reader_role
```

To revoke the USAGE privilege on the sequence generator `order_id` from the role `sales_role`, use the following syntax:

```
REVOKE USAGE ON SEQUENCE order_id FROM sales_role;
```

To revoke the USAGE privilege on the user-defined type `price` from the role `finance_role`, use the following syntax:

```
REVOKE USAGE ON TYPE price FROM finance_role;
```

To revoke the USAGE privilege on the user-defined aggregate `types.maxPrice` from the role `sales_role`, use the following syntax:

```
REVOKE USAGE ON DERBY AGGREGATE types.maxPrice FROM sales_role;
```

SELECT statement

The SELECT statement performs a query on one or more tables.

Syntax

```
query
[ ORDER BY clause ]
[ result offset clause ]
[ fetch first clause ]
[ FOR UPDATE clause ]
[ WITH { RR | RS | CS | UR } ]
```

A SELECT statement consists of a query with an optional ORDER BY clause, an optional result offset clause, an optional fetch first clause, an optional FOR UPDATE clause, and an optional isolation level. The SELECT statement is so named because the typical first word of the query construct is SELECT. (A *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 result offset clause and the fetch first clause can be used to fetch only a subset of the otherwise selected rows, possibly with an offset into the result set. The FOR UPDATE clause makes the result set's cursor updatable. The SELECT statement supports the FOR FETCH ONLY clause. The FOR FETCH ONLY clause is synonymous with the FOR READ ONLY clause.

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

For queries that do not select a specific column from the tables involved in the SELECT statement (for example, queries that use `COUNT(*)`), the user must have at least one column-level SELECT privilege or table-level SELECT privilege. See GRANT statement for more information.

Example

```
-- 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* have an underlying associated SQL cursor, often referred to as the result set's cursor. The cursor can be 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. The FOR UPDATE clause can be used to ensure a compilation check that the SELECT statement meets the requiremments of a updatable cursors, or to limit the columns that can be updated.
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.

Requirements for updatable cursors and updatable ResultSets
Only simple, single-table SELECT cursors 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
  - ORDER BY 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
- If the underlying *query* has a WHERE clause, the WHERE clause must not have subqueries.

Note: Cursors are read-only by default. To produce an updatable cursor besides meeting the requirements listed above, the concurrency mode for the ResultSet must be `ResultSet.CONCUR_UPDATABLE` or the SELECT statement must have FOR UPDATE in the FOR clause (see FOR UPDATE clause).

There is no SQL language statement to *assign* a name to a cursor. Instead, one can 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 the *Derby Developer's Guide*.

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.

SET statements

Use the SET statements to set the current deferrability for constraints or to set the current role, schema, or isolation level.

SET CONSTRAINTS statement

The SET CONSTRAINTS statement sets the deferrability of one or more constraints.

The SET CONSTRAINTS statement allows you to set the constraint mode for one or more constraints either to DEFERRED or to IMMEDIATE.

When you use the statement to change a constraint from DEFERRED to IMMEDIATE, the constraint is checked as soon as the statement is executed.

If the check fails, the transaction is not rolled back; an error here constitutes a statement level error only. Therefore, you can use this statement to check if all constraints are fulfilled before you attempt to commit the transaction.

A SET CONSTRAINTS statement changes the state of a constraint only until the transaction ends (or until another, overriding SET CONSTRAINTS statement is issued). Once the transaction ends, the constraint reverts to the default behavior declared for it at the time it was created (using a CREATE TABLE or ALTER TABLE statement).

For more information on deferrable constraints, see CONSTRAINT clause and constraintCharacteristics.

It is recommended that you use SET CONSTRAINTS on table-level constraints. If you use SET CONSTRAINTS on a column-level constraint, you will need to find the name of the corresponding index by performing queries against the system tables, which is cumbersome and requires additional non-portable SQL.

Note: The SET CONSTRAINTS statement is valid only after a database has been fully upgraded to Derby Release 10.11 or higher. (See "Upgrading a database" in the *Derby Developer's Guide* for more information.) This statement has no meaning in a database that is at Release 10.10 or lower.

Syntax

```
SET CONSTRAINTS constraintNameList { DEFERRED | IMMEDIATE }
```

The *constraintNameList* is defined as follows:

```
ALL | constraintName [ { , constraintName }... ]
```

Runtime behavior

If the constraint mode is DEFERRED and a violation is seen at commit time, an exception is thrown, and the transaction is rolled back.

When you change the constraint mode explicitly to IMMEDIATE using SET CONSTRAINTS, the constraint is checked, but slightly differently from the way it is checked at commit time: if a violation is found, a statement-level exception is thrown. You can use this behavior to verify that constraints are fulfilled before you attempt to commit.

If the constraint mode is IMMEDIATE upon entering a stored routine, and that routine in a nested connection changes the constraint mode to DEFERRED, any constraints that are affected are checked upon return from the routine. If the check fails, an exception is thrown, and the transaction is rolled back.

Constraints with a constraint mode of DEFERRED are also checked if the application calls *XAResource.prepare(Xid)*. If there is a violation, Derby throws *XAException.XA_RBINTEGRITY*, and the XA transaction is rolled back.

Examples

```
SET CONSTRAINTS FOO DEFERRED;

SET CONSTRAINTS ALL DEFERRED;

SET CONSTRAINTS FOO, BAR IMMEDIATE;
```

SET ISOLATION statement

The SET ISOLATION statement changes the isolation level for a user's connection.

Valid isolation levels are SERIALIZABLE, REPEATABLE READ, READ COMMITTED, and READ UNCOMMITTED.

Issuing this statement always commits the current transaction. The JDBC *java.sql.Connection.setTransactionIsolation* method behaves almost identically to this command, with one exception: if you are using the embedded driver, and if the call to *java.sql.Connection.setTransactionIsolation* does not actually change the isolation level (that is, if it sets the isolation level to its current value), the current transaction is not committed.

For information about isolation levels, see "Locking, concurrency, and isolation" in the *Derby Developer's Guide*. For information about the JDBC *java.sql.Connection.setTransactionIsolation* method, see java.sql.Connection.setTransactionIsolation method.

Syntax

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

Example

```
set isolation serializable;
```

SET ROLE statement

The SET ROLE statement sets the current role for the current SQL context of a session.

You can set a role only if the current user has been granted the role, or if the role has been granted to PUBLIC.

For more information on roles, see "Using SQL roles" in the *Derby Security Guide*.

Syntax

```
SET ROLE { roleName | 'stringConstant' | ? | NONE }
```

If you specify a *roleName* of NONE, the effect is to unset the current role.

If you specify the role as a string constant or as a dynamic parameter specification (?), any leading and trailing blanks are trimmed from the string before attempting to use the remaining (sub)string as a *roleName*. The dynamic parameter specification can be used in prepared statements, so the SET ROLE statement can be prepared once and then executed with different role values. You cannot specify NONE as a dynamic parameter.

Setting a role identifies a set of privileges that is a union of the following:

- The privileges granted to that role
- The union of privileges of roles contained in that role (for a definition of role containment, see "Syntax for roles" in GRANT statement)

In a session, the *current privileges* define what the session is allowed to access. The *current privileges* are the union of the following:

- The privileges granted to the current user
- The privileges granted to PUBLIC
- The privileges identified by the current role, if set

The SET ROLE statement is not transactional; a rollback does not undo the effect of setting a role. If a transaction is in progress, an attempt to set a role results in an error.

Examples

```
SET ROLE reader;
```

```
  // These examples show the use of SET ROLE in JDBC statements.
  // The case normal form is visible in the SYS.SYSROLES system table.
  stmt.execute("SET ROLE admin");      -- case normal form: ADMIN
  stmt.execute("SET ROLE \"admin\"");  -- case normal form: admin
  stmt.execute("SET ROLE none");       -- special case

  PreparedStatement ps = conn.prepareStatement("SET ROLE ?");
  ps.setString(1, "  admin ");  -- on execute: case normal form: ADMIN
  ps.setString(1, "\"admin\""); -- on execute: case normal form: admin
  ps.setString(1, "none");      -- on execute: syntax error
  ps.setString(1, "\"none\"");  -- on execute: case normal form: none
```

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 | ? | 'stringConstant' }
  |
SET CURRENT SQLID [ = ] { schemaName | USER | ? | 'stringConstant' }
```

The *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, and sys.)

USER is the current user. If no current user is defined, the current schema defaults to 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".

Examples

```
-- 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
```

TRUNCATE TABLE statement

The TRUNCATE TABLE statement quickly removes all content from the specified table and returns it to its initial empty state.

To truncate a table, you must be either the database owner or the table owner.

You cannot truncate system tables or global temporary tables with this statement.

Syntax

```
TRUNCATE TABLE tableName
```

Examples
To truncate the entire `Flights` table, use the following statement:

```
TRUNCATE TABLE Flights;
```

UPDATE statement

The UPDATE statement updates the value of one or more columns of a table.

Syntax

```
{
    UPDATE tableName [ [ AS ] correlationName ] ]
        SET columnName = value
        [ , columnName = value ]*
        [ WHERE clause ]
  |
    UPDATE tableName
        SET columnName = value
        [ , columnName = value ]*
        WHERE CURRENT OF
}
```

where *value* is defined as follows:

```
expression | DEFAULT
```

The first syntactical form, called a searched update, updates the value of one or more columns for all rows of the table for which the WHERE clause evaluates to TRUE.

The second syntactical form, called a positioned update, updates one or more columns on the current row of an open, updatable cursor. If columns were specified in the FOR UPDATE clause of the SELECT statement used to generate the cursor, only those columns can be updated. If no columns were specified or the select statement did not include a FOR UPDATE clause, all columns may be updated.

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

The DEFAULT literal is the only value which you can directly assign to a generated column. Whenever you alter the value of a column referenced by the *generationClause* of a generated column, Derby recalculates the value of the generated column.

Example

```
-- 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 employees without a specific job title to
 MANAGER
UPDATE EMPLOYEE
 SET JOB = 'MANAGER'
 WHERE JOB IS NULL;
// 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.

SQL clauses

CONSTRAINT clause

A CONSTRAINT clause is an optional part of a CREATE TABLE statement or an ALTER TABLE statement. A constraint is a rule to which data must conform. Constraint names are optional.

See CREATE TABLE statement and rrefsqlj81859 for details on those statements.

A CONSTRAINT can be one of the following:

- A *columnLevelConstraint*

  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 *tableLevelConstraint*

  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.
- FOREIGN KEY

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

- CHECK

  Specifies rules for values in the column.

Table constraints include:

- PRIMARY KEY

  Specifies the column or columns that uniquely identify a row in the table. NULL values are not allowed.
- UNIQUE

  Specifies that values in the columns must be unique.
- FOREIGN KEY

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

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

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

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

A constraint operates with the privileges of the owner of the constraint. See "Using SQL standard authorization" and "Privileges on views, triggers, and constraints" in the *Derby Developer's Guide* for details.

Deferrable constraints

Constraints can be *deferred*, meaning that Derby does not check constraints immediately. By default, a constraint is checked as soon as a statement completes. Deferrable constraints allow temporary breaches of constraints for more flexible insert and update operations.

Note: Deferrable constraints are available only after a database has been fully upgraded to Derby Release 10.11 or higher. (See "Upgrading a database" in the *Derby Developer's Guide* for more information.) They cannot be used in a database that is at Release 10.10 or lower.

When a deferrable constraint's constraint mode is DEFERRED before execution of a statement starts, the checking of the constraint does not take place at the end of the statement execution as usual, but only when it is explicitly or implicitly requested using one of the following mechanisms:

- The transaction ends (a commit operation takes place)
- A SET CONSTRAINTS statement which sets the constraint mode to IMMEDIATE is executed
- A return from a stored procedure or function reverts the constraint mode to IMMEDIATE

The point at which a deferrable constraint is checked is referred to as the *deferred checking time*.

If the constraint mode of a constraint is IMMEDIATE before a call to a stored procedure or function, and the stored procedure or function sets the constraint mode of that

constraint to DEFERRED, the constraint mode is implicitly reset to IMMEDIATE on return from the stored procedure. This happens because the constraint mode is pushed on a stack when we enter the stored procedure or function (as are other session state variables, like the current role). If a constraint violation happens as a result, the transaction is rolled back and an exception is thrown.

See Referential actions for information about the behavior of deferrable foreign keys.

Primary key 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.

Unique constraints

A UNIQUE constraint defines a set of columns that uniquely identify rows in a table only if all the key values are not NULL. If one or more key parts are NULL, duplicate keys are allowed.

For example, if there is a UNIQUE constraint on `col1` and `col2` of a table, the combination of the values held by `col1` and `col2` will be unique as long as these values are not NULL. If one of `col1` and `col2` holds a NULL value, there can be another identical row in the table.

A table 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*.)

If the REFERENCES clause contains a CASCADE or SET NULL referential action, the primary or unique key referenced must not be deferrable.

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 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.

If the constraint mode is IMMEDIATE (the default), Derby performs constraint checks at the time the statement is executed. If the constraint mode is DEFERRED, the checking is done later, typically at commit time. See Deferrable constraints for more information.

Backing indexes

UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints generate indexes that enforce or "back" the constraint (and are sometimes called *backing indexes*). PRIMARY KEY constraints generate unique indexes. FOREIGN KEY constraints generate non-unique indexes. UNIQUE constraints generate unique indexes if all the columns are non-nullable, and they generate non-unique indexes if one or more columns are nullable. 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 conditions

If a check constraint is specified as part of a *columnDefinition*, 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 statement is rolled back.
- If the update rule is NO ACTION, Derby checks the dependent tables for foreign key constraints *after* all updates and BEFORE triggers have been executed, but *before* AFTER 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 as part of a CASCADE on the parent table, 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 and BEFORE triggers have been executed, but *before* AFTER 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 statement 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.

If ON DELETE is not specified, NO ACTION is the implicit delete rule.

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.

If a foreign key's constraint mode is DEFERRED, an insert (or update of a row that changes the foreign key) in the child table will be checked at deferred checking time, notwithstanding the ON DELETE or ON UPDATE referential action specification. If a row in the parent table is deleted (or updated so as to modify the referenced key), the behavior depends on the specification of ON DELETE or ON UPDATE. Only if NO ACTION has been specified is the checking ever deferred. If the primary table's referenced primary or unique key constraint is also deferred, any delete of a parent row can lead to a foreign key violation immediately (or at deferred checking time, if the foreign key is also deferred, as the case may be) when the last of possibly several key duplicates of the referenced key is deleted or updated.

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.

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)
 );

-- use the same check constraint, but
-- make the MEAL_CONSTRAINT deferrable
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'))
     DEFERRABLE INITIALLY DEFERRED,
 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
 );
```

columnLevelConstraint

```
[ CONSTRAINT constraintName ]
{
    NOT NULL |
    CHECK ( searchCondition ) |
```

```
    PRIMARY KEY |
    UNIQUE |
    REFERENCES clause
} [ constraintCharacteristics ]
```

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

If a *constraintName* is not specified, Derby generates a unique constraint name.

tableLevelConstraint

```
[ CONSTRAINT constraintName ]
{
    CHECK ( searchCondition ) |
    {
        PRIMARY KEY ( simpleColumnName [ , simpleColumnName ]* ) |
        UNIQUE ( simpleColumnName [ , simpleColumnName ]* ) |
        FOREIGN KEY ( simpleColumnName
 [ , simpleColumnName ]*
 )
REFERENCES clause
    }
} [ constraintCharacteristics ]
```

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

If a *constraintName* is not specified, Derby generates a unique constraint name.

REFERENCES clause

```
REFERENCES tableName [ ( simpleColumnName [ , simpleColumnName ]* ) ]
[ 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 } ]
```

constraintCharacteristics

```
constraintCheckTime [ [ NOT ] DEFERRABLE ] |
[ NOT ] DEFERRABLE [ constraintCheckTime ]
```

The *constraintCheckTime* is defined as follows:

```
INITIALLY DEFERRED | INITIALLY IMMEDIATE
```

If DEFERRABLE is specified, the constraint is deferrable; otherwise it is not deferrable unless INITIALLY DEFERRED is specified. To make a constraint from an existing database deferrable, you must drop and recreate the constraint.

If *constraintCheckTime* is not specified, INITIALLY IMMEDIATE is implicit.

If INITIALLY DEFERRED is specified and DEFERRABLE is not specified, DEFERRABLE is implicit. If INITIALLY DEFERRED is specified, NOT DEFERRABLE is not permitted.

The deferrability or the *constraintCheckTime* (that is, the default checking time) of a constraint cannot be altered. To change these characteristics, you must drop the constraint and recreate it.

NOT NULL constraints are not deferrable; all others are deferrable. The NOT NULL constraint can, however, be dropped and recreated if desired. This will require a full table scan.

A constraint can be specified as DEFERRABLE or NOT DEFERRABLE, or with a *constraintCheckTime* of INITIALLY DEFERRED or INITIALLY IMMEDIATE, only after a database has been fully upgraded to Derby Release 10.11 or higher. (See "Upgrading a

database" in the *Derby Developer's Guide* for more information.) These keywords have no meaning in a database that is at Release 10.10 or lower.

After a full upgrade to Release 10.11 or higher, old constraints on the database will be converted to NOT DEFERRABLE, the default value.

Note: Deferred constraints sometimes impose extra performance overhead to allow for the deferred checking. If your application does not require deferred checking, we recommend that you make constraints NOT DEFERRABLE (the default).
Note: In contrast to constraint checking, the referential actions specified by a referential constraint are never deferred. In Derby, these actions are RESTRICT, SET NULL and CASCADE for delete and RESTRICT for update. If NO ACTION is specified, the referential check can be deferred.

EXTERNAL NAME clause

The EXTERNAL NAME clause specifies the Java method to be called in a CREATE FUNCTION or CREATE PROCEDURE statement, and it specifies a Java class in a CREATE AGGREGATE or CREATE TYPE statement.

See CREATE FUNCTION statement, CREATE PROCEDURE statement, CREATE DERBY AGGREGATE statement, and CREATE TYPE statement for more information.

Syntax

```
EXTERNAL NAME singleQuotedString
```

The *singleQuotedString* cannot have any extraneous spaces.

The method name specified in the CREATE FUNCTION or CREATE PROCEDURE statement normally takes the following form:

```
'class_name.method_name[(parameterTypes)]'
```

The optional *parameterTypes* specification is needed when the Java signature determined from the SQL declaration is ambiguous.

If the class is a static nested class, or if the method is in a static nested class, use a dollar sign ($) as a separator between the outer and static class. For example, suppose you have the following class and method definition:

```
public class TestFuncs {
    public static final class MyMath {
        public static double pow( double base, double power ) {
            return Math.pow( base, power );
        }
    }
}
```

If you use CREATE FUNCTION to bind this *pow* method to a user-defined function, the external name should be *TestFuncs$MyMath.pow*, not *TestFuncs.MyMath.pow*.

Examples

```
-- Specify the Mode class as an external name
CREATE DERBY AGGREGATE mode FOR INT
EXTERNAL NAME 'com.example.myapp.aggs.Mode';
```

```
-- Specify the pow method in the static class MyMath
CREATE FUNCTION MYPOWER ( X DOUBLE, Y DOUBLE )
RETURNS DOUBLE
PARAMETER STYLE JAVA
NO SQL LANGUAGE JAVA
EXTERNAL NAME 'TestFuncs$MyMath.pow'
```

```
-- create a function to round a Double to a specified number of decimal
 places
CREATE FUNCTION DoubleFormat (value FLOAT, places INTEGER)
RETURNS FLOAT
PARAMETER STYLE JAVA
LANGUAGE JAVA
EXTERNAL NAME
'utils.Utils.Double(java.lang.Float,java.lang.Integer)'
```

FOR UPDATE clause

The FOR UPDATE clause is an optional part of a SELECT statement.

Cursors are read-only by default. The FOR UPDATE clause specifies that the cursor should be updatable, and enforces a check during compilation that the SELECT statement meets the requirements for an updatable cursor. For more information about updatability, see Requirements for updatable cursors and updatable ResultSets.

Syntax

```
FOR
{
    READ ONLY |
    FETCH ONLY |
    UPDATE [ OF simpleColumnName [ , simpleColumnName ]* ]
}
```

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

Instead of FOR UPDATE, you can specify FOR READ ONLY or its synonym, FOR FETCH ONLY, to indicate that the result set is not updatable.

Note: The use of the FOR UPDATE clause is not mandatory to obtain an updatable JDBC ResultSet. As long as the statement used to generate the JDBC ResultSet meets the requirements for updatable cursor, it is sufficient for the JDBC Statement that generates the JDBC ResultSet to have concurrency mode `ResultSet.CONCUR_UPDATABLE` for the ResultSet to be updatable.

The optimizer is able to use an index even if the column in the index is being updated.

For information about how indexes affect performance, see *Tuning Derby*.

Example

```
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. See *selectExpression* for more information.

Syntax

```
FROM tableExpression [ , tableExpression ]*
```

Examples

```
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 join operation. 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. See *selectExpression* for more information.

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

Using the ROLLUP syntax, you can specify that multiple levels of grouping should be computed at once.

Syntax

```
GROUP BY
{
    columnName [ , columnName ]* |
    ROLLUP ( columnName [ , columnName ]* )
}
```

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

The *selectItems* in the *selectExpression* with a GROUP BY clause must contain only aggregates or grouping columns.

Examples

```
-- 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_name), 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

-- Compute sub-totals of Sales_History data, grouping it by Region, by
-- (Region, State), and by (Region, State, Product), as well as computing
-- an overall total of the sales for all Regions, States, and Products:
SELECT Region, State, Product, SUM(Sales) Total_Sales
FROM Sales_History
GROUP BY ROLLUP(Region, State, Product)
```

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.

See *selectExpression* for more information.

Syntax

```
HAVING booleanExpression
```

The *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.

Example

```
-- 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
```

WINDOW clause

The WINDOW clause allows you to refer to a window by name when you use a ROW NUMBER function in a *selectExpression*.

See ROW_NUMBER function and selectExpression for more information.

Syntax

```
WINDOW windowName AS windowSpecification
```

In a WINDOW clause, *windowName* is a *SQLIdentifier*.

Currently, the only valid *windowSpecification* is a set of empty parentheses (( )), which indicates that the function is evaluated over the entire result set.

Example

```
SELECT ROW_NUMBER() OVER R,
    B,
    SUM(A)
FROM T5 GROUP BY B WINDOW R AS ()
```

ORDER BY clause

The ORDER BY clause is an optional element of several statements, expressions, and subqueries.

It can be an element of the following:

- A SELECT statement
- A *selectExpression*
- A VALUES expression
- A *scalarSubquery*
- A *tableSubquery*

It can also be used in an INSERT statement or a CREATE VIEW statement.

An ORDER BY clause allows you to specify the order in which rows appear in the result set. In subqueries, the ORDER BY clause is meaningless unless it is accompanied by one or both of the result offset and fetch first clauses or in conjunction with the ROW_NUMBER function, since there is no guarantee that the order is retained in the outer result set. It is permissible to combine ORDER BY on the outer query with ORDER BY in subqueries.

Syntax

```
ORDER BY { columnName | columnPosition | expression }
    [ ASC | DESC ]
    [ NULLS FIRST | NULLS LAST ]
    [ , columnName | columnPosition | expression
        [ ASC | DESC ]
        [ NULLS FIRST | NULLS LAST ]
    ]*
```

columnName
: Refers to the names visible from the *selectItems* in the underlying query of the SELECT statement. The *columnName* that you specify in the ORDER BY clause does not need to be the SELECT list.

columnPosition
: An integer that identifies the number of the column in the *selectItems* in the underlying query of the SELECT statement. The *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 specified in the SELECT list.

expression
: A sort key expression, such as numeric, string, and datetime expressions. An *expression* can also be a row value expression such as a *scalarSubquery* or case expression.

ASC
: Specifies that the results should be returned in ascending order. If the order is not specified, ASC is the default.

DESC

Specifies that the results should be returned in descending order.

NULLS FIRST

Specifies that NULL values should be returned before non-NULL values.

NULLS LAST

Specifies that NULL values should be returned after non-NULL values.

Notes

- If SELECT DISTINCT is specified or if the SELECT statement contains a GROUP BY clause, the ORDER BY columns must be in the SELECT list.
- An ORDER BY clause prevents a SELECT statement from being an updatable cursor. For more information, see Requirements for updatable cursors and updatable ResultSets.
- If the null ordering is not specified then the handling of the null values is:
  - NULLS LAST if the sort is ASC
  - NULLS FIRST if the sort is DESC
- If neither ascending nor descending order is specified, and the null ordering is also not specified, then both defaults are used and thus the order will be ascending with NULLS LAST.

Example using a correlation name

You can sort the result set by a correlation name, if the correlation name is specified in the select list. For example, to return from the CITIES database all of the entries in the CITY_NAME and COUNTRY columns, where the COUNTRY column has the correlation name NATION, you specify this SELECT statement:

```
SELECT CITY_NAME, COUNTRY AS NATION
    FROM CITIES
    ORDER BY NATION
```

Example using a numeric expression

You can sort the result set by a numeric expression, for example:

```
SELECT name, salary, bonus FROM employee
   ORDER BY salary+bonus
```

In this example, the salary and bonus columns are DECIMAL data types.

Example using a function

You can sort the result set by invoking a function, for example:

```
SELECT i, len FROM measures
   ORDER BY sin(i)
```

Example specifying null ordering

You can specify the position of NULL values using the null ordering specification:

```
SELECT * FROM t1 ORDER BY c1 DESC NULLS LAST
```

The result offset and fetch first clauses

The *result offset clause* provides a way to skip the first *N* rows in a result set before starting to return any rows. The *fetch first clause*, which can be combined with the *result offset clause* if desired, limits the number of rows returned in the result set.

The *fetch first clause* can sometimes be useful for retrieving only a few rows from an otherwise large result set, usually in combination with an ORDER BY clause. The use of this clause can give efficiency benefits. In addition, it can make programming the application simpler.

Syntax

```
OFFSET { integerLiteral | ? } { ROW | ROWS }
```

```
FETCH { FIRST | NEXT } [ integerLiteral | ? ] { ROW | ROWS } ONLY
```

ROW is synonymous with ROWS and FIRST is synonymous with NEXT.

For the *result offset clause*, the value of the integer literal (or the dynamic parameter ?) must be equal to 0 (default if the clause is not given), or positive. If it is larger than the number of rows in the underlying result set, no rows are returned.

For the *fetch first clause*, the value of the literal (or the dynamic parameter ?) must be 1 or higher. The literal can be omitted, in which case it defaults to 1. If the clause is omitted entirely, all rows (or those rows remaining if a *result offset clause* is also given) will be returned.

Examples

```
-- Fetch the first row of T
SELECT * FROM T FETCH FIRST ROW ONLY

-- Sort T using column I, then fetch rows 11 through 20 of the sorted
--   rows (inclusive)
SELECT * FROM T ORDER BY I OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY

-- Skip the first 100 rows of T
-- If the table has fewer than 101 records, an empty result set is
--   returned
SELECT * FROM T OFFSET 100 ROWS

-- Use of ORDER BY and FETCH FIRST in a subquery
SELECT DISTINCT A.ORIG_AIRPORT, B.FLIGHT_ID FROM
   (SELECT FLIGHT_ID, ORIG_AIRPORT
       FROM FLIGHTS
       ORDER BY ORIG_AIRPORT DESC
       FETCH FIRST 40 ROWS ONLY)
    AS A, FLIGHTAVAILABILITY AS B
   WHERE A.FLIGHT_ID = B.FLIGHT_ID

JDBC (using a dynamic parameter):
PreparedStatement p =
    con.prepareStatement("SELECT * FROM T ORDER BY I OFFSET ? ROWS");
p.setInt(1, 100);
ResultSet rs = p.executeQuery();
```

Note: Make sure to specify the ORDER BY clause if you expect to retrieve a sorted result set. If you do not use an ORDER BY clause, the result set that is retrieved will typically have the order in which the records were inserted.

USING clause

The USING clause specifies which columns to test for equality when two tables are joined.

It can be used instead of an ON clause in the JOIN operations that have an explicit join clause.

Syntax

```
USING ( simpleColumnName [ , simpleColumnName ]* )
```

The columns listed in the USING clause must be present in both of the two tables being joined. The USING clause will be transformed to an ON clause that checks for equality between the named columns in the two tables.

When a USING clause is specified, an asterisk (*) in the select list of the query will be expanded to the following list of columns (in this order):

- All the columns in the USING clause
- All the columns of the first (left) table that are not specified in the USING clause
- All the columns of the second (right) table that are not specified in the USING clause

An asterisk qualified by a table name (for example, COUNTRIES.*) will be expanded to every column of that table that is not listed in the USING clause.

If a column in the USING clause is referenced without being qualified by a table name, the column reference points to the column in the first (left) table if the join is an INNER JOIN or a LEFT OUTER JOIN. If it is a RIGHT OUTER JOIN, unqualified references to a column in the USING clause point to the column in the second (right) table.

Examples

The following query performs an inner join between the COUNTRIES table and the CITIES table on the condition that COUNTRIES.COUNTRY is equal to CITIES.COUNTRY:

```
SELECT * FROM COUNTRIES JOIN CITIES
     USING (COUNTRY)
```

The next query is similar to the one above, but it has the additional join condition that COUNTRIES.COUNTRY_ISO_CODE is equal to CITIES.COUNTRY_ISO_CODE:

```
SELECT * FROM COUNTRIES JOIN CITIES
    USING (COUNTRY, COUNTRY_ISO_CODE)
```

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 *selectExpression* 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 booleanExpression
```

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

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

Example

```
-- 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.

Updatable and/or scrollable JDBC *ResultSet*s can provide a simpler and easier way to perform these tasks.

See UPDATE statement and DELETE statement for more information on those statements. For more information about updatable cursors, see SELECT statement. For information on scrollable and updatable *ResultSet*s, see the Java SE API documentation on the *java.sql.ResultSet* interface as well as the information on the Derby implementation at java.sql.ResultSet interface.

Syntax

```
WHERE CURRENT OF cursorName
```

Example

```
conn.setAutoCommit(false);

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

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. If not otherwise specified, an expression is permitted anywhere the word *expression* appears in the syntax. This includes:

- ORDER BY clause
- *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.

The following tables list all the possible SQL expressions and indicate where the expressions are allowed.

General expressions

General expressions are expressions that might result in a value of any type. The following table lists the types of general expressions.

Table 4. General expressions

Expression Type	Explanation
Column reference	A *columnName* that references the value of the column made visible to the expression containing the Column reference. You must qualify the *columnName* by the table name or correlation name if it is ambiguous. The qualifier of a *columnName* must be the correlation name, if a correlation name is given to a table that is in a FROM clause. The table name is no longer visible as a *columnName* qualifier once it has been aliased by a correlation name. Allowed in *selectExpression*s, UPDATE statements, and the WHERE clauses of data manipulation statements.
Constant	Most built-in data types typically have constants associated with them (as shown in Data types).
NULL	NULL is an untyped constant representing the unknown value. 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.
Dynamic parameter	A dynamic parameter is a parameter to an SQL statement for which the value is not specified when the statement is created. Instead, the statement has a question mark (?) as a placeholder for each dynamic parameter. See Dynamic parameters. 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. Allowed anywhere in an expression where the data type can be easily deduced. See Dynamic parameters.
CAST expression	Lets you specify the type of NULL or of a dynamic parameter or convert a value to another type. See CAST function.

Expression Type	Explanation
Scalar subquery	Subquery that returns a single row with a single column. See *scalarSubquery*.
Table subquery	Subquery that returns more than one column and more than one row. See *tableSubquery*. Allowed as a *tableExpression* in a FROM clause and with EXISTS, IN, and quantified comparisons.
Conditional expression	A conditional expression chooses an expression to evaluate based on a boolean test. Conditional expressions include the CASE expression, the NULLIF function, and the COALESCE function.

Boolean expressions

Boolean expressions are expressions that result in boolean values. Most general expressions can result in boolean values. See Boolean expressions for more information and a table of operators.

Numeric expressions

Numeric expressions are expressions that result in numeric values. Most of the general expressions can result in numeric values. Numeric values have one of the following types:

- BIGINT
- DECIMAL
- DOUBLE PRECISION
- INTEGER
- REAL
- SMALLINT

The following table lists the types of numeric expressions.

Table 5. Numeric expressions

Expression Type	Explanation
+, -, *, /, unary + and - expressions	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. See AVG function.
SUM	Returns the sum of a set of numeric values. See SUM function.
LENGTH	Returns the number of characters in a character or bit string. See LENGTH function.
LOWER	See LCASE or LOWER function.

Expression Type	Explanation
COUNT	Returns the count of a set of values. See COUNT function, COUNT(*) function.

Character expressions

Character expressions are expressions that result in a CHAR or VARCHAR value. Most general expressions can result in a CHAR or VARCHAR value. The following table lists the types of character expressions.

Table 6. Character expressions

Expression Type	Explanation
A CHAR or VARCHAR value that uses wildcards.	The wildcards % and _ make a character string a pattern against which the LIKE operator can look for 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 operator.
Built-in string functions	The built-in string functions act on a String and return a string. See LTRIM function, LCASE or LOWER function, RTRIM function, TRIM function, SUBSTR function, and UCASE or UPPER function.
USER functions	User functions return information about the current user as a String. See CURRENT_USER function, SESSION_USER function, and USER function.

Date and time expressions

A date or time expression results in a DATE, TIME, or TIMESTAMP value. Most of the general expressions can result in a date or time value. The following table lists the types of date and time expressions.

Table 7. Date and time expressions

Expression Type	Explanation
CURRENT_DATE	Returns the current date. See CURRENT_DATE function.
CURRENT_TIME	Returns the current time. See CURRENT_TIME function.
CURRENT_TIMESTAMP	Returns the current timestamp. See CURRENT_TIMESTAMP function.

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 ]
[ WINDOW clause ]
[ ORDER BY clause ]
[ result offset clause ]
[ fetch first clause ]
```

selectItem:

```
{
    * |
    { tableName | correlationName } .* |
    expression [AS simpleColumnName]
}
```

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

A *selectItem* projects one or more result column values for a table result being constructed in a *selectExpression*.

For queries that do not select a specific column from the tables involved in the *selectExpression* (for example, queries that use `COUNT(*)`), the user must have at least one column-level SELECT privilege or table-level SELECT privilege. See GRANT statement for more information.

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 COUNT(*),WORK_DEPT,AVG(SALARY)
     FROM EMPLOYEE
     GROUP BY WORK_DEPT
     ORDER BY 3
```

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.

The WINDOW clause allows you to refer to a window by name when you use a ROW_NUMBER function in a *selectExpression*.

The ORDER BY clause allows you to specify the order in which rows appear in the result set. In subqueries, the ORDER BY clause is meaningless unless it is accompanied by one or both of the result offset and fetch first clauses or in conjunction with the ROW_NUMBER function.

The result offset clause provides a way to skip the N first rows in a result set before starting to return any rows. The fetch first clause, which can be combined with the result offset clause if desired, limits the number of rows returned in the result set.

Derby processes a *selectExpression* in the following order:

- FROM clause
- WHERE clause
- GROUP BY (or implicit GROUP BY)
- HAVING clause
- WINDOW clause
- ORDER BY clause
- Result offset clause
- Fetch first 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 expression, 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. 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.

Examples

```
-- 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
```

tableExpression

A *tableExpression* specifies a table, view, or function in a FROM clause.

A *tableExpression* 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.

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

Syntax

```
{
    tableViewOrFunctionExpression |
    joinOperation
}
```

Example

```
-- 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
```

tableViewOrFunctionExpression

```
{
    { tableName | viewName }
    [ correlationClause ]
|
    { tableSubquery | tableFunctionInvocation }
    correlationClause
}
```

where *correlationClause* is

```
[ AS ]
correlationName
[ ( simpleColumnName [ , simpleColumnName ]* ) ]
```

tableFunctionInvocation:

```
TABLE functionName( [ [ functionArg ] [, functionArg ]* ] )
```

Note that when you invoke a table function, you must bind it to a correlation name. For example:

```
SELECT s.*
FROM TABLE( externalEmployees( 42 ) ) s
```

NEXT VALUE FOR expression

The NEXT VALUE FOR expression retrieves the next value from a sequence generator.

A sequence generator is created with a CREATE SEQUENCE statement.

Syntax

```
NEXT VALUE FOR sequenceName
```

If this is the first use of the sequence generator, the generator returns its START value. Otherwise, the INCREMENT value is added to the previous value returned by the sequence generator. The data type of the value is the *dataType* specified for the sequence generator.

If the sequence generator wraps around, then one of the following happens:

- If the sequence generator was created using the CYCLE keyword, the sequence generator is reset to its START value.
- If the sequence generator was created with the default NO CYCLE behavior, Derby throws an exception.

In order to retrieve the next value of a sequence generator, you or your session's current role must have USAGE privilege on the generator.

A NEXT VALUE FOR expression may occur in the following places:

- SELECT statement: As part of the expression defining a returned column in a SELECT list
- VALUES expression: As part of the expression defining a column in a row constructor (VALUES expression)
- UPDATE statement; As part of the expression defining the new value to which a column is being set

Only one NEXT VALUE FOR expression is allowed per sequence per statement.

The NEXT VALUE FOR expression is not allowed in any statement which has a DISTINCT or ORDER BY expression.

The next value of a sequence generator is not affected by whether the user commits or rolls back a transaction which invoked the sequence generator.

A NEXT VALUE expression may not appear in any of these situations:

- CASE expression
- WHERE clause
- ORDER BY clause
- Aggregate expression
- ROW_NUMBER function
- DISTINCT select list

Examples

```
VALUES (NEXT VALUE FOR order_id);
```

```
INSERT INTO re_order_table
  SELECT NEXT VALUE FOR order_id, order_date, quantity
  FROM orders
  WHERE back_order = 1;
```

```
UPDATE orders
  SET oid = NEXT VALUE FOR order_id
  WHERE expired = 1;
```

VALUES expression

The VALUES expression allows construction of a row or a table from other values.

A VALUES expression can be used in all the places where a query can, and thus can be used in any of the following ways:

- As a statement that returns a *ResultSet*
- Within expressions and statements wherever subqueries are permitted
- As the source of values for an INSERT statement (in an INSERT statement, you normally use a VALUES expression when you do not use a *selectExpression*)

Syntax

```
{
    VALUES ( value [ , value ]* )
        [ , ( value [ , value ]* ) ]*
  |
    VALUES value [ , value ]*
} [ ORDER BY clause ]
  [ result offset clause ]
  [ fetch first clause ]
```

where *value* is defined as

```
expression | DEFAULT
```

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

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

A VALUES expression that is used in an INSERT statement cannot use an ORDER BY, result offset, or fetch first clause. However, if the VALUES expression does not contain the DEFAULT keyword, the VALUES clause can be put in a subquery and ordered, as in the following statement:

```
INSERT INTO t SELECT * FROM (VALUES 'a','c','b') t ORDER BY 1;
```

Examples

```
-- 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)
-- using ORDER BY and FETCH FIRST
VALUES (3,21),(1,22),(2,23) ORDER BY 1 FETCH FIRST 2 ROWS ONLY
-- using ORDER BY and OFFSET
VALUES (3,21),(1,22),(2,23) ORDER BY 1 OFFSET 1 ROW
-- 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)
```

Expression precedence

Precedence of operations from highest to lowest is as follows.

- ( ), ?, constants (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 expressions

Boolean expressions are expressions that result in boolean values.

Most of the expressions listed in the table General expressions can result in boolean values.

Boolean expressions are allowed in the following clauses and operations:

- WHERE clause
- Check constraints (boolean expressions in check constraints have limitations; see CONSTRAINT clause for details)
- CASE expression
- HAVING clause (with restrictions)
- ON clauses of INNER JOIN, LEFT OUTER JOIN, and RIGHT OUTER JOIN operations

A boolean expression can include a boolean operator or operators. These operators are listed in the following table.

Table 8. 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` **Note:** Derby also accepts the != operator, which is not included in the SQL standard.	`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_'`	`characterExpression [ NOT ] LIKE characterExpressionWithWildCard [ ESCAPE 'escapeCharacter'] `

Operator	Explanation and Example	Syntax
	To treat % or _ as constant characters, escape the character with an optional escape character, which you specify with the ESCAPE clause. `SELECT a FROM tabA WHERE a LIKE '%=_' ESCAPE '='` **Note:** When LIKE comparisons are used, Derby compares one character at a time for non-metacharacters. This is different than the way Derby processes = comparisons. The comparisons with the = operator compare the entire character string on left side of the = operator with the entire character string on the right side of the = operator. For more information, see "Character-based collation in Derby" in the *Derby Developer's Guide*.	
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. A table subquery can return multiple columns and rows. `WHERE EXISTS (SELECT * FROM Flights WHERE dest_airport = 'SFO' AND orig_airport = 'GRU')`	**[NOT] EXISTS** ***tableSubquery***

Operator	Explanation and Example	Syntax

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 subquery. ANY and SOME are equivalent.
	
	`WHERE normal_rate < ALL (SELECT budget/550 FROM Groups)`
		expression *comparisonOperator*
		{
		ALL |
		ANY |
		SOME
		}
		tableSubquery

CASE expression

The CASE expression can be used for conditional expressions in Derby.

See SQL expressions for more information on expressions.

Syntax

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

Derby supports three kinds of CASE expressions, which we refer to as a searched CASE expression, a simple CASE expression, and an extended CASE expression.

The syntax of a searched CASE expression is as follows:

```
CASE
  WHEN booleanExpression THEN thenExpression
  [ WHEN booleanExpression THEN thenExpression ]*
  [ ELSE elseExpression ]
END
```

The syntax of a simple CASE expression is as follows:

```
CASE valueExpression
  WHEN valueExpression [ , valueExpression ]* THEN thenExpression
  [ WHEN valueExpression [ , valueExpression ]* THEN thenExpression ]*
  [ ELSE elseExpression ]
END
```

A *valueExpression* is an expression that resolves to a single value.

For both searched and simple CASE expressions, both *thenExpression* and *elseExpression* are defined as follows:

```
NULL | valueExpression
```

The *thenExpression* and *elseExpression* must be type-compatible. For built-in types, this means that the types must be the same or that a built-in broadening conversion must exist between the types.

The syntax of an extended CASE expression is as follows:

```
CASE valueExpression
  WHEN whenOperand [ , whenOperand ]* THEN thenExpression
  [ WHEN whenOperand [ , whenOperand ]* THEN thenExpression ]*
  [ ELSE elseExpression ]
END
```

A *whenOperand* is defined as follows:

```
valueExpression
 |
comparisonOperatorexpression |
IS [ NOT ] NULL |
[ NOT ] LIKE characterExpressionWithWildCard [ ESCAPE 'escapeCharacter' ]
 |
[ NOT ] BETWEEN expression AND expression |
[ NOT ] IN tableSubquery |
[ NOT ] IN ( expression [, expression
 ]* )
 |
comparisonOperator { ALL | ANY | SOME } tableSubquery
```

A *comparisonOperator* is defined as follows:

```
{ < | = | > | <= | >= | <> }
```

For details on LIKE expressions, see Boolean expressions.

For all types of CASE expressions, if an ELSE clause is not specified, ELSE NULL is implicit.

Example

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

-- simple CASE expression, equivalent to previous expression
-- returns 3
VALUES CASE 1 WHEN 1 THEN 3 ELSE 4 END

-- searched CASE expression
-- returns 7
VALUES
   CASE
      WHEN 1 = 2 THEN 3
      WHEN 4 = 5 THEN 6
      ELSE 7
   END

-- simple CASE expression
-- returns 'two'
VALUES
  CASE 1+1
    WHEN 1 THEN 'one'
    WHEN 2 THEN 'two'
    ELSE 'many'
  END

-- simple CASE expression
-- returns 'odd', 'even', 'big'
SELECT
  CASE X
    WHEN 1, 3, 5, 7, 9 THEN 'odd'
    WHEN 2, 4, 6, 8, 10 THEN 'even'
    ELSE 'big'
  END
FROM
```

```
  (VALUES 5, 8, 12) AS V(X)

-- extended CASE expression
-- returns ('long', 182), ('medium', 340), ('short', 20)
SELECT DISTANCE, COUNT(*)
FROM (SELECT
        CASE MILES
          WHEN < 250 THEN 'short'
          WHEN BETWEEN 250 AND 2000 THEN 'medium'
          WHEN > 2000 THEN 'long'
        END
      FROM FLIGHTS) AS F(DISTANCE)
GROUP BY DISTANCE
```

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 question mark (?).

The JDBC API documentation refers to dynamic parameters as IN, INOUT, or OUT parameters. In SQL, they are always IN parameters.

You must specify values for dynamic parameters 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 DATE
   ```
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 DATE
   ```
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 operator 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.

```
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. Dynamic parameters are allowed as the operand of the unary operators - or +. For example:

```
CREATE TABLE t1 (c11 INT, c12 SMALLINT, c13 DOUBLE, c14 CHAR(3))
SELECT * FROM t1 WHERE c11 BETWEEN -? AND +?
-- The type of both of the unary operators is INT
-- based on the context in which they are used (that is,
-- because c11 is INT, the unary parameters also get the
-- type INT.
```

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 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.

JOIN operations

The JOIN operations perform joins between two tables.

The JOIN operations are among the possible *tableExpression*s in a FROM clause. (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`.)

The JOIN operations are:

- INNER JOIN operation

  Specifies a join between two tables with an explicit join clause.
- LEFT OUTER JOIN operation

  Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the first table.
- RIGHT OUTER JOIN operation

  Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the second table.

- CROSS JOIN operation

  Specifies a join that produces the Cartesian product of two tables. It has no explicit join clause.
- NATURAL JOIN operation

  Specifies an inner or outer join between two tables. It has no explicit join clause. Instead, one is created implicitly using the common columns from the two tables.

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*.

INNER JOIN operation

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

Syntax

```
tableExpression [ INNER ] JOIN tableExpression
{
    ON booleanExpression |
    USING clause
}
```

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).

Examples

```
-- 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
```

LEFT OUTER JOIN operation

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

Syntax

```
tableExpression LEFT [ OUTER ] JOIN tableExpression
{
    ON booleanExpression |
    USING clause
}
```

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).

Example 1

```
-- match cities to countries in Asia

SELECT CITIES.COUNTRY, CITIES.CITY_NAME, REGION
FROM Countries
LEFT OUTER JOIN Cities
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE REGION = 'Asia'

-- use the synonymous syntax, LEFT JOIN, to achieve exactly
-- the same results as in the example above

SELECT  COUNTRIES.COUNTRY, CITIES.CITY_NAME,REGION
FROM COUNTRIES
LEFT JOIN CITIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE REGION = 'Asia'
```

Example 2

```
-- 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
```

RIGHT OUTER JOIN operation

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

B LEFT OUTER JOIN A is equivalent to A RIGHT OUTER JOIN B, with the columns in a different order.

Syntax

```
tableExpression RIGHT [ OUTER ] JOIN tableExpression
{
    ON booleanExpression |
    USING clause
}
```

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).

Example 1

```
-- get all countries and corresponding cities, including
-- countries without any cities

SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME
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 COUNTRIES.COUNTRY, CITIES.CITY_NAME
FROM CITIES
RIGHT OUTER JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE Countries.region = 'Africa'

-- use the synonymous syntax, RIGHT JOIN, to achieve exactly
-- the same results as in the example above
```

```
SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME
FROM CITIES
RIGHT JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE Countries.region = 'Africa'
```

Example 2

```
-- a tableExpression can be a join operation. 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
```

CROSS JOIN operation

A CROSS JOIN is a JOIN operation that produces the Cartesian product of two tables. Unlike other JOIN operators, it does not let you specify a join clause. You may, however, specify a WHERE clause in the SELECT statement.

Syntax

```
tableExpression CROSS JOIN
{
    tableViewOrFunctionExpression |
    ( tableExpression )
}
```

Examples

The following SELECT statements are equivalent:

```
SELECT * FROM CITIES CROSS JOIN FLIGHTS
```

```
SELECT * FROM CITIES, FLIGHTS
```

The following SELECT statements are equivalent:

```
SELECT * FROM CITIES CROSS JOIN FLIGHTS
    WHERE CITIES.AIRPORT = FLIGHTS.ORIG_AIRPORT
```

```
SELECT * FROM CITIES INNER JOIN FLIGHTS
    ON CITIES.AIRPORT = FLIGHTS.ORIG_AIRPORT
```

The following example is more complex. The ON clause in this example is associated with the LEFT OUTER JOIN operation. Note that you can use parentheses around a JOIN operation.

```
SELECT * FROM CITIES LEFT OUTER JOIN
    (FLIGHTS CROSS JOIN COUNTRIES)
        ON CITIES.AIRPORT = FLIGHTS.ORIG_AIRPORT
            WHERE COUNTRIES.COUNTRY_ISO_CODE = 'US'
```

A CROSS JOIN operation can be replaced with an INNER JOIN where the join clause always evaluates to true (for example, 1=1). It can also be replaced with a sub-query. So equivalent queries would be:

```
SELECT * FROM CITIES LEFT OUTER JOIN
    FLIGHTS INNER JOIN COUNTRIES ON 1=1
        ON CITIES.AIRPORT = FLIGHTS.ORIG_AIRPORT
            WHERE COUNTRIES.COUNTRY_ISO_CODE = 'US'
```

```
SELECT * FROM CITIES LEFT OUTER JOIN
    (SELECT * FROM FLIGHTS, COUNTRIES) S
        ON CITIES.AIRPORT = S.ORIG_AIRPORT
            WHERE S.COUNTRY_ISO_CODE = 'US'
```

NATURAL JOIN operation

A NATURAL JOIN is a JOIN operation that creates an implicit join clause for you based on the common columns in the two tables being joined. Common columns are columns that have the same name in both tables.

A NATURAL JOIN can be an INNER join, a LEFT OUTER join, or a RIGHT OUTER join. The default is INNER join.

If the SELECT statement in which the NATURAL JOIN operation appears has an asterisk (*) in the select list, the asterisk will be expanded to the following list of columns (in this order):

- All the common columns
- Every column in the first (left) table that is not a common column
- Every column in the second (right) table that is not a common column

An asterisk qualified by a table name (for example, COUNTRIES.*) will be expanded to every column of that table that is not a common column.

If a common column is referenced without being qualified by a table name, the column reference points to the column in the first (left) table if the join is an INNER JOIN or a LEFT OUTER JOIN. If it is a RIGHT OUTER JOIN, unqualified references to a common column point to the column in the second (right) table.

Syntax

```
tableExpression NATURAL [ { LEFT | RIGHT } [ OUTER ] | INNER ] JOIN
{
    tableViewOrFunctionExpression |
    ( tableExpression )
}
```

Examples

If the tables COUNTRIES and CITIES have two common columns named COUNTRY and COUNTRY_ISO_CODE, the following two SELECT statements are equivalent:

```
SELECT * FROM COUNTRIES NATURAL JOIN CITIES
```

```
SELECT * FROM COUNTRIES JOIN CITIES
    USING (COUNTRY, COUNTRY_ISO_CODE)
```

The following example is similar to the one above, but it also preserves unmatched rows from the first (left) table:

```
SELECT * FROM COUNTRIES NATURAL LEFT JOIN CITIES
```

SQL queries

query

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

Syntax

```
{
```

```
    ( query
         [ ORDER BY clause ]
         [ result offset clause ]
         [ fetch first clause ]
    ) |
    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)

-- Use of ORDER BY and FETCH FIRST in a subquery
SELECT DISTINCT A.ORIG_AIRPORT, B.FLIGHT_ID FROM
   (SELECT FLIGHT_ID, ORIG_AIRPORT
```

```
      FROM FLIGHTS
      ORDER BY ORIG_AIRPORT DESC
      FETCH FIRST 40 ROWS ONLY)
   AS A, FLIGHTAVAILABILITY AS B
  WHERE A.FLIGHT_ID = B.FLIGHT_ID

-- 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')
```

scalarSubquery

A *scalarSubquery*, sometimes called an expression subquery, is a subquery that evaluates to a single row with a single column.

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.

Syntax

```
( query
    [ ORDER BY clause ]
    [ result offset clause ]
    [ fetch first clause ]
)
```

Examples

```
-- 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
-- 1 column "R1" and 1 row of data.
SELECT R1
 FROM (VALUES('GROUP 1')) AS X(R1)
```

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, or with EXISTS, it can return multiple columns.

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

Syntax

```
( query
    [ ORDER BY clause ]
    [ result offset clause ]
    [ fetch first clause ]
)
```

Example

```
-- a subquery used as a tableExpression in a FROM clause
SELECT VirtualFlightTable.flight_ID
FROM
    (SELECT flight_ID, orig_airport, dest_airport
    FROM Flights
    WHERE (orig_airport = 'SFO' OR dest_airport = 'SCL') )
AS VirtualFlightTable
-- a subquery (values expression) used as a tableExpression
-- in a FROM clause
SELECT mycol1
FROM
    (VALUES (1, 2), (3, 4))
AS mytable (mycol1, mycol2)
-- a subquery used with EXISTS
SELECT *
FROM Flights
WHERE EXISTS
    (SELECT * FROM Flights WHERE dest_airport = 'SFO'
    AND orig_airport = 'GRU')
-- a subquery used with IN
SELECT flight_id, segment_number
FROM Flights
WHERE flight_id IN
    (SELECT flight_ID
    FROM Flights WHERE orig_airport = 'SFO'
```

```
    OR dest_airport = 'SCL')
-- a subquery with ORDER BY and FETCH FIRST clauses
SELECT flight_id, segment_number
FROM Flights
WHERE flight_id IN
    (SELECT flight_ID
    FROM Flights WHERE orig_airport = 'SFO'
    OR dest_airport = 'SCL' ORDER BY flight_id FETCH FIRST 12 ROWS ONLY)
-- a subquery used with a quantified comparison
SELECT NAME, COMM
FROM STAFF
WHERE COMM >
(SELECT AVG(BONUS + 800)
     FROM EMPLOYEE
     WHERE COMM < 5000)
```

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 *SQLIdentifier*s 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

The standard built-in functions supported in Derby are as follows.

- ABS or ABSVAL function
- ACOS function
- ASIN function
- ATAN function
- ATAN2 function
- BIGINT function
- CAST function
- CEIL or CEILING function
- CHAR function
- Concatenation operator
- COS function
- NULLIF function
- CURRENT_DATE function
- CURRENT ISOLATION function
- CURRENT_TIME function
- CURRENT_TIMESTAMP function
- CURRENT_USER function
- DATE function
- DAY function
- DEGREES function
- DOUBLE function
- EXP function
- FLOOR function
- HOUR function
- IDENTITY_VAL_LOCAL function
- INTEGER function
- LENGTH function
- LN or LOG function
- LOG10 function

- LOCATE function
- LCASE or LOWER function
- LTRIM function
- MINUTE function
- MOD function
- MONTH function
- PI function
- RADIANS function
- RTRIM function
- SECOND function
- SESSION_USER function
- SIN function
- SMALLINT function
- SQRT function
- SUBSTR function
- TAN function
- TIME function
- TIMESTAMP function
- TRIM function
- UCASE or UPPER function
- USER function
- VARCHAR function
- YEAR function

Aggregates (set functions)

This section describes aggregates (also described as *set functions* in ANSI SQL and as *column functions* in some database literature).

Aggregates 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 count rows as well as calculate the minimum, maximum, sum, count, average, variance, and standard deviation of an expression over a set of values.

In addition to the built-in aggregates, Derby allows you to create custom aggregate operators, called user-defined aggregates (UDAs). For information on creating and removing UDAs, see CREATE DERBY AGGREGATE statement and DROP DERBY AGGREGATE statement. See GRANT statement and REVOKE statement for information on usage privileges for UDAs.

For information on writing the Java classes that implement UDAs, see "Programming user-defined aggregates" in the *Derby Developer's Guide*.

The built-in aggregates can operate on expressions that evaluate to the data types shown in the following table.

Table 9. Permitted data types for built-in aggregates

Function Name	Permitted Data Types
AVG	Numeric built-in data types
COUNT	All types
MAX	Data types that can be indexed
MIN	Data types that can be indexed
STDDEV_POP	Numeric built-in data types

Function Name	Permitted Data Types
STDDEV_SAMP	Numeric built-in data types
SUM	Numeric built-in data types
VAR_POP	Numeric built-in data types
VAR_SAMP	Numeric built-in data types

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.

ABS or ABSVAL function

The ABS or ABSVAL function returns the absolute value of a numeric expression.

The return type is the type of the parameter. All built-in numeric types are supported (DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, BIGINT, NUMERIC, REAL, and SMALLINT).

Syntax

```
ABS ( numericExpression )
```

```
ABSVAL ( numericExpression )
```

Example

```
-- returns 3
```

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

ACOS function

The ACOS function returns the arc cosine of a specified number.

The specified number is the cosine, in radians, of the angle that you want. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the absolute value of the specified number is greater than 1, an exception is returned that indicates that the value is out of range (SQL state 22003).

The returned value, in radians, is in the range of zero (0) to pi. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
ACOS ( number )
```

ASIN function

The ASIN function returns the arc sine of a specified number.

The specified number is the sine, in radians, of the angle that you want. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.
- If the absolute value of the specified number is greater than 1, an exception is returned that indicates that the value is out of range (SQL state 22003).

The returned value, in radians, is in the range -pi/2 to pi/2. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
ASIN ( number )
```

ATAN function

The ATAN function returns the arc tangent of a specified number.

The specified number is the tangent, in radians, of the angle that you want. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The returned value, in radians, is in the range -pi/2 to pi/2. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
ATAN ( number )
```

ATAN2 function

The ATAN2 function returns the arctangent, in radians, of the quotient of the two arguments.

Upon successful completion, the function returns the arc tangent of *y*/*x* in the range *-pi* to *pi* radians, where *y* is the first argument and *x* is the second argument. The specified numbers must be DOUBLE PRECISION numbers.

- If either argument is NULL, the result of the function is NULL.
- If the first argument is zero and the second argument is positive, the result of the function is zero.
- If the first argument is zero and the second argument is negative, the result of the function is the double value closest to *pi*.
- If the first argument is positive and the second argument is zero, the result is the double value closest to *pi*/2.
- If the first argument is negative and the second argument is zero, the result is the double value closest to *-pi*/2.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
ATAN2 ( y, x )
```

AVG function

AVG is an aggregate function that evaluates the average of an expression over a set of rows.

AVG is allowed only on expressions that evaluate to numeric data types.

See Aggregates (set functions) for more information about these functions.

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 numeric data type. You can therefore call methods that evaluate to SQL 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 function

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.

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

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

CAST function

The CAST function 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 data types

The following table shows valid 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 data types. The first row lists the target data types. A "Y" indicates that a conversion from the source to the target is valid. For example, the first cell in the second row lists the source data type SMALLINT. The remaining cells on the second row indicate the whether or not you can convert SMALLINT to the target data types that are listed in the first row of the table.

Table 10. Explicit conversions between source types and target types for SQL data types

Types	BOOLEAN	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	XML
BOOLEAN	Y	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-	-
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	Y	-
VARCHAR	Y	Y	Y	Y	Y	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y	-
LONG VARCHAR	Y	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-	-
CHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-	-
VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-	-
LONG VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-	-
CLOB	Y	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-	-

Types	BOOLEAN	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	XML
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-
DATE	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-	-
TIME	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-	-
TIMESTAMP	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	Y	Y	-
XML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	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 data types are categorized as follows:

- *logical*
  - BOOLEAN
- *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 to and from logical types

A BOOLEAN value can be cast explicitly to any of the string types. The result is 'true', 'false', or null. Conversely, string types can be cast to BOOLEAN. However, an error

is raised if the string value is not 'true', 'false', 'unknown', or null. Casting 'unknown' to boolean results in a null value.

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 to 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 cast to and from other types: you can cast a BLOB only to another BLOB, but you can cast other bit string types to a BLOB.

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.

Conversions of XML values

An XML value cannot be converted to any non-XML type using an explicit or implicit CAST. Use the XMLSERIALIZE operator to convert an XML type to a character type.

Examples

```
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)
```

CEIL or CEILING function

The CEIL or CEILING function rounds the specified number up, and returns the smallest number that is greater than or equal to the specified number.

The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the returned value is NULL.
- If the specified number is equal to a mathematical integer, the returned value is the same as the specified number.
- If the specified number is zero (0), the returned value is zero.
- If the specified number is less than zero but greater than -1.0, the returned value is zero.

The returned value is the smallest (closest to negative infinity) double floating-point value that is greater than or equal to the specified number. The returned value is equal to a mathematical integer. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
CEIL ( number )
```

```
CEILING ( number )
```

CHAR function

The CHAR function returns a fixed-length character string representation.

The representations are:

- 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 CHAR 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. The first argument cannot be an XML value. To convert an XML value to a CHAR of a specified length, you must use the SQL/XML serialization operator XMLSERIALIZE.

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 *characterExpression* 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 *characterExpression* 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 *characterExpression* 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. The result is left justified.

- If the first argument is a SMALLINT: 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 an 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 BIGINT: 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.

Floating point to character syntax

```
CHAR ( floatingPointExpression )
```

floatingPointExpression

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

Example

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).

COALESCE function

The COALESCE function takes two or more compatible arguments and returns the first argument that is not null.

The result is null only if all the arguments are null.

If all the parameters of the function call are dynamic, an error occurs.

Note: A synonym for COALESCE is VALUE. VALUE is accepted by Derby but is not recognized by the SQL standard.

Syntax

```
COALESCE ( expression, expression [ , expression ]* )
```

The function must have at least two arguments.

Example

```
ij> -- create table with three different integer types
ij> create table temp(smallintcol smallint, bigintcol bigint, intcol
 integer);
0 rows inserted/updated/deleted

ij> insert into temp values (1, null, null);
1 row inserted/updated/deleted
ij> insert into temp values (null, 2, null);
1 row inserted/updated/deleted
ij> insert into temp values (null, null, 3);
1 row inserted/updated/deleted

ij> select * from temp;
SMALL&|BIGINTCOL           |INTCOL
--------------------------------------
1     |NULL                |NULL
NULL  |2                   |NULL
NULL  |NULL                |3

3 rows selected

ij> -- the return data type of coalesce is bigint
ij> select coalesce (smallintcol, bigintcol) from temp;
1
--------------------
1
2
NULL

3 rows selected

ij> -- the return data type of coalesce is bigint
ij> select coalesce (smallintcol, bigintcol, intcol) from temp;
1
--------------------
1
2
3

3 rows selected

ij> -- the return data type of coalesce is integer
ij> select coalesce (smallintcol, intcol) from temp;
1
-----------
1
NULL
3

3 rows selected
```

Concatenation operator

The concatenation operator, | |, concatenates its right operand to the end of its left operand. It operates on character or bit expressions.

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.

Examples

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

COS function

The COS function returns the cosine of a specified number.

The specified number is the angle, in radians, that you want the cosine for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.

Syntax

```
COS ( number )
```

COSH function

The COSH function returns the hyperbolic cosine of a specified number.

The specified number is the angle, in radians, that you want the hyperbolic cosine for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is one (1.0).

Syntax

```
COSH ( number )
```

COT function

The COT function returns the cotangent of a specified number.

The specified number is the angle, in radians, that you want the cotangent for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.

Syntax

```
COT ( number )
```

COUNT function

COUNT is an aggregate function that counts the number of rows accessed in an expression. COUNT is allowed on all types of expressions.

See Aggregates (set functions) for more information about these functions.

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 INTEGER.

Example

```
-- 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(*) function

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.

See Aggregates (set functions) for more information about these functions.

Syntax

```
COUNT(*)
```

The resulting data type is INTEGER.

Example

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

CURRENT DATE function

CURRENT DATE is a synonym for CURRENT_DATE.

See CURRENT_DATE function for details.

CURRENT_DATE function

The CURRENT_DATE function 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
```

Example

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

CURRENT ISOLATION function

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

Syntax

```
CURRENT ISOLATION
```

Example

```
VALUES CURRENT ISOLATION
```

CURRENT_ROLE function

The CURRENT_ROLE function returns the authorization identifier of the current role. If there is no current role, it returns NULL.

This function returns a string of up to 258 characters. This is twice the length of an identifier (128*2) + 2, to allow for quoting.

Syntax

```
CURRENT_ROLE
```

Example

```
VALUES CURRENT_ROLE
```

CURRENT SCHEMA function

The CURRENT SCHEMA function 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
```

Example

```
-- 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 function

CURRENT TIME is a synonym for CURRENT_TIME.

See CURRENT_TIME function for details.

CURRENT_TIME function

The CURRENT_TIME function 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
```

Examples

```
VALUES CURRENT_TIME

-- or, alternately:

VALUES CURRENT TIME
```

CURRENT TIMESTAMP function

CURRENT TIMESTAMP is a synonym for CURRENT_TIMESTAMP.

See CURRENT_TIMESTAMP function for details.

CURRENT_TIMESTAMP function

The CURRENT_TIMESTAMP function 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
```

Examples

```
VALUES CURRENT_TIMESTAMP

-- or, alternately:

VALUES CURRENT TIMESTAMP
```

CURRENT_USER function

When used outside stored routines, the CURRENT_USER, USER, and SESSION_USER functions all return the authorization identifier of the user that created the SQL session.

See USER function and SESSION_USER function for details on those functions.

SESSION_USER also always returns this value when used within stored routines.

If used within a stored routine created with EXTERNAL SECURITY DEFINER, however, CURRENT_USER and USER return the authorization identifier of the user that owns the schema of the routine. This is usually the creating user, although the database owner could be the creator as well.

For information about definer's and invoker's rights, see CREATE PROCEDURE statement or CREATE FUNCTION statement.

These functions return a string of up to 128 characters.

Syntax

```
CURRENT_USER
```

Example

```
VALUES CURRENT_USER
```

DATE function

The DATE function returns a date from a value.

The argument must be a date, timestamp, a positive number less than or equal to 2,932,897, a valid string representation of a date or timestamp, or a string of length 7 that is not a CLOB, LONG VARCHAR, or XML value. 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, 1970, 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 )
```

Examples

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

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

This example results in an internal representation of '1972-02-28'.

```
VALUES DATE(789)
```

DAY function

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 not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 1 and 31. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
DAY ( expression )
```

Example

```
values day('2007-08-02');
```

The resulting value is 2.

DEGREES function

The DEGREES function converts a specified number from radians to degrees.

The specified number is an angle measured in radians, which is converted to an approximately equivalent angle measured in degrees. The specified number must be a DOUBLE PRECISION number.

Attention: The conversion from radians to degrees is not exact. You should not expect DEGREES(ACOS(0.5)) to return exactly 60.0.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
DEGREES ( number )
```

DOUBLE function

The DOUBLE function returns a floating-point number corresponding to a number or a character string.

The returned value corresponds to a number if the argument is a numeric expression.

The returned value corresponds to a 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.

EXP function

The EXP function returns e raised to the power of the specified number.

The specified number is the exponent that you want to raise e to. The specified number must be a DOUBLE PRECISION number.

The constant e is the base of the natural logarithms.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
EXP ( number )
```

FLOOR function

The FLOOR function rounds the specified number down, and returns the largest number that is less than or equal to the specified number.

The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is equal to a mathematical integer, the result of this function is the same as the specified number.
- If the specified number is zero (0), the result of this function is zero.

The returned value is the largest (closest to positive infinity) double floating point value that is less than or equal to the specified number. The returned value is equal to a mathematical integer. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
FLOOR ( number )
```

HOUR function

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 not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 0 and 24. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
HOUR ( expression )
```

Example

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 function

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 or a single row UPDATE statement.

Syntax

```
IDENTITY_VAL_LOCAL ( )
```

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 or UPDATE statement. The INSERT statement must contain a VALUES clause on a table containing an identity column. The function returns a null value when a single row UPDATE statement or 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 or single row UPDATE statement for a table without an identity column
- A multiple row INSERT statement with a VALUES clause
- A multiple row UPDATE statement
- 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, or an UPDATE trigger defined that updates 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
ij> update t1 set c1=default where c2=4;
1 row inserted/updated/deleted
ij> values IDENTITY_VAL_LOCAL();
1
------------------------------
9
1 row selected
ij> select * from t1;
C1         |C2
----------------------
1          |8
2          |1
3          |8
4          |9
5          |1
6          |2
7          |3
9          |4
8 rows selected
ij> update t1 set c1=default where c2=8;
2 rows inserted/updated/deleted
ij> values IDENTITY_VAL_LOCAL();
1
------------------------------
9
1 row selected
ij> select * from t1;
C1         |C2
----------------------
10         |8
2          |1
11         |8
4          |9
5          |1
6          |2
```

```
7           |3
9           |4
```

INTEGER function

The INTEGER function returns an integer representation of a number or character string 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 *numericExpression*, 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 *characterExpression*, 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.

Example
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
```

LCASE or LOWER function

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

Syntax

```
LCASE ( characterExpression )
```

```
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.

Examples

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

SELECT LOWER(flight_id) FROM Flights
```

LENGTH function

The LENGTH function 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 )
```

Examples

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

LN or LOG function

The LN and LOG functions return the natural logarithm (base e) of the specified number.

The specified number must be a DOUBLE PRECISION number that is greater than zero (0).

- If the specified number is NULL, the result of these functions is NULL.
- If the specified number is zero or a negative number, an exception is returned that indicates that the value is out of range (SQL state 22003).

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
LN ( number )
```

```
LOG ( number )
```

LOG10 function

The LOG10 function returns the base-10 logarithm of the specified number.

The specified number must be a DOUBLE PRECISION number that is greater than zero (0).

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero or a negative number, an exception is returned that indicates that the value is out of range (SQL state 22003).

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
LOG10 ( number )
```

LOCATE function

The LOCATE function is used to search for a string within another string. If the desired string is found, LOCATE returns the index at which it is found. If the desired string is not found, LOCATE returns 0.

Syntax

```
LOCATE ( characterExpression, characterExpression [ , startPosition ] )
```

There are two required arguments to the LOCATE function, and a third optional argument.

- The first *characterExpression* specifies the string to search **for**.
- The second *characterExpression* specifies the string **in which** to search.
- The third argument is the *startPosition*, which specifies the position in the second argument at which the search is to start. If the third argument is not provided, the LOCATE function starts its search at the beginning of the second argument.

The return type for LOCATE is an integer. The LOCATE function returns an integer indicating the index position within the second argument at which the first argument was first located. Index positions start with 1. If the first argument is not found in the second argument, LOCATE returns 0. If the first argument is an empty string (`''`), LOCATE returns the value of the third argument (or 1 if it was not provided), even if the second argument is also an empty string. If a NULL value is passed for either of the *characterExpression* arguments, NULL is returned.

Examples

```
-- returns 2, since 'love' is found at index position 2:
VALUES LOCATE('love', 'clover')

-- returns 0, since 'stove' is not found in 'clover':
VALUES LOCATE('stove', 'clover')

-- returns 5 (note the start position is 4):
VALUES LOCATE('iss', 'Mississippi', 4)

-- returns 1, because the empty string is a special case:
VALUES LOCATE('', 'ABC')

-- returns 0, because 'AAA' is not found in '':
VALUES LOCATE('AAA', '')

-- returns 3
VALUES LOCATE('', '', 3)
```

LTRIM function

The LTRIM function 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.

Example

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

MAX function

MAX is an aggregate function that evaluates the maximum of an expression over a set of rows.

See Aggregates (set functions) for more information about these functions.

MAX is allowed only on expressions that evaluate to indexable data types (specifically, those marked with a Y in the second table, "Comparisons allowed by Derby", in Data type assignments and comparison, sorting, and ordering). This means that MAX cannot be used with expressions that evaluate to BLOB, CLOB, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, XML, or user-defined types.

Syntax

```
MAX ( [ DISTINCT | ALL ] expression )
```

The DISTINCT and ALL qualifiers eliminate or retain duplicates, but 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.

The type's comparison rules determine the maximum value. For CHAR and 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 blank spaces are ignored for character comparisons.

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

Examples

```
-- 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 function

MIN is an aggregate function that evaluates the minimum of an expression over a set of rows.

See Aggregates (set functions) for more information about these functions.

MIN is allowed only on expressions that evaluate to indexable data types (specifically, those marked with a Y in the second table, "Comparisons allowed by Derby", in Data type assignments and comparison, sorting, and ordering). This means that MIN cannot be used with expressions that evaluate to BLOB, CLOB, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, XML, or user-defined types.

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 minimum value. For CHAR and VARCHAR, the number of blank spaces at the end of the value can affect how MIN 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 minimum, because blank spaces are ignored for character comparisons.

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

Examples

```
-- 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 function

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 not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 0 and 59. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
MINUTE ( expression )
```

Example

Select all rows from the "flights" table where the "departure_time" is between 6:00 and 6:30 AM:

```
SELECT * FROM flights
   WHERE HOUR(departure_time) = 6 and MINUTE(departure_time) < 31;
```

MOD function

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

Syntax

```
MOD ( integerExpression, integerExpression )
```

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 function

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 not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 1 and 12. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
MONTH ( expression )
```

Example

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

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

NULLIF function

The NULLIF function can be used for conditional expressions in Derby.

Syntax

```
NULLIF ( expression, expression )
```

The NULLIF function is very similar to the CASE expression. It returns NULL if the two arguments are equal, and it returns the first argument if they are not equal. For example,

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

is equivalent to the following CASE expression:

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

PI function

The PI function returns a value that is closer than any other value to pi.

The constant pi is the ratio of the circumference of a circle to the diameter of a circle.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
PI ( )
```

RADIANS function

The RADIANS function converts a specified number from degrees to radians.

The specified number is an angle measured in degrees, which is converted to an approximately equivalent angle measured in radians. The specified number must be a DOUBLE PRECISION number.

Attention: The conversion from degrees to radians is not exact.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
RADIANS ( number )
```

RANDOM function

The RANDOM function returns a random number.

The RANDOM function returns a DOUBLE PRECISION number with positive sign, greater than or equal to zero (0), and less than one (1.0).

Syntax

```
RANDOM ( )
```

RAND function

The RAND function returns a random number given a seed number

The RAND function returns a DOUBLE PRECISION number with positive sign, greater than or equal to zero (0), and less than one (1.0), given an INTEGER seed number.

Syntax

```
RAND ( seed )
```

ROW_NUMBER function

The ROW_NUMBER function returns the row number over a named or unnamed window specification.

The ROW_NUMBER function does not take any arguments, and for each row over the window it returns an ever increasing BIGINT. It is normally used to limit the number of rows returned for a query. A result offset or fetch first clause can be a more efficient way to perform this task.

The data type of the returned value is BIGINT.

Syntax

```
ROW_NUMBER ( ) OVER [ windowSpecification | windowName ]
```

Currently, the only valid *windowSpecification* is an empty pair of parentheses (( )), which indicates that the function is evaluated over the entire result set.

If you choose to use a WINDOW clause in a *selectExpression* to specify a window, you must specify a *windowName* to refer to it.

Examples

To limit the number of rows returned from a query to the first 10 rows of table T, use the following query:

```
SELECT * FROM (
   SELECT
```

```
     ROW_NUMBER() OVER () AS R,
     T.*
   FROM T
) AS TR
   WHERE R <= 10;
```

To display the result of a query using a window name in a WINDOW clause:

```
SELECT ROW_NUMBER() OVER R,
    B,
    SUM(A)
FROM T5 GROUP BY B WINDOW R AS ()
```

RTRIM function

The RTRIM function 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.

Examples

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

SECOND function

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 not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 0 and 59. If the argument can be null, the result can be null. If the argument is null, the result is 0.

Syntax

```
SECOND ( expression )
```

Example
The RECEIVED column contains a timestamp that has an internal value equivalent to 2005-12-25-17.12.30.000000. To return only the seconds part of the timestamp, use the following syntax:

```
SECOND(RECEIVED)
```

The value 30 is returned.

SESSION_USER function

When used outside stored routines, the SESSION_USER, USER, and CURRENT_USER functions all return the authorization identifier of the user that created the SQL session.

See USER function and CURRENT_USER function for details on those functions.

SESSION_USER also always returns this value when used within stored routines.

If used within a stored routine created with EXTERNAL SECURITY DEFINER, however, USER and CURRENT_USER return the authorization identifier of the user that owns the schema of the routine. This is usually the creating user, although the database owner could be the creator as well.

For information about definer's and invoker's rights, see CREATE PROCEDURE statement or CREATE FUNCTION statement.

Syntax

```
SESSION_USER
```

Example

```
VALUES SESSION_USER
```

SIGN function

The SIGN function returns the sign of the specified number.

The specified number is the number you want the sign of. The specified number must be a DOUBLE PRECISION number.

The data type of the returned value is INTEGER.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero (0).
- If the specified number is greater than zero (0), the result of this function is plus one (+1).
- If the specified number is less than zero (0), the result of this function is minus one (-1).

Syntax

```
SIGN ( number )
```

SIN function

The SIN function returns the sine of a specified number.

The specified number is the angle, in radians, that you want the sine for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
SIN ( number )
```

SINH function

The SINH function returns the hyperbolic sine of a specified number.

The specified number is the angle, in radians, that you want the hyperbolic sine for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
SINH ( number )
```

SMALLINT function

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.

Examples

To determine the small integer representation of the number 32767.99, use this clause:

```
VALUES SMALLINT (32767.99)
```

The result is 32767.

To determine the small integer representation of the number 1, use this clause:

```
VALUES SMALLINT (1)
```

The result is 1.

SQRT function

The SQRT function 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 ( number )
```

Examples

```
-- 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 function

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 a 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 ] )
```

The parameter *startPosition* and the optional parameter *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.

The parameter *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, the *startPosition* and *lengthOfString* parameters refer to characters. For bit expressions, the *startPosition* and *lengthOfString* parameters refer to bits.

If the *startPosition* is positive, it refers to position from the start of the source expression (counting the first character as 1). The *startPosition* cannot be a negative number.

If the *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*. The SUBSTR function returns an error if you specify a negative number for the parameter *lengthOfString*.

Examples
To return a substring of the word `hello`, starting at the second character and continuing until the end of the word, use the following clause:

```
VALUES SUBSTR('hello', 2)
```

The result is '`ello`'.

To return a substring of the word `hello`, starting at the first character and continuing for two characters, use the following clause:

```
VALUES SUBSTR('hello',1,2)
```

The result is 'he'.

STDDEV_POP function

STDDEV_POP is an aggregate function that evaluates the standard deviation of an expression over a population.

See Aggregates (set functions) for more information about these functions.

STDDEV_POP is allowed only on expressions that evaluate to numeric data types.

Syntax

```
STDDEV_POP ( expression )
```

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 DOUBLE (it might overflow).

Formula

Standard deviation is calculated as follows:

```
sqrt( var_pop() )
```

Example

```
-- find the standard deviation in flight time per aircraft
SELECT AIRCRAFT, STDDEV_POP( flying_time )
FROM flights GROUP BY aircraft;
```

STDDEV_SAMP function

STDDEV_SAMP is an aggregate function that evaluates the sample deviation of an expression over a set of rows.

See Aggregates (set functions) for more information about these functions.

STDDEV_SAMP is allowed only on expressions that evaluate to numeric data types.

Syntax

```
STDDEV_SAMP ( expression )
```

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 DOUBLE (it might overflow).

Formula

Sample deviation is calculated as follows:

```
sqrt( var_samp() )
```

Example

```
-- find the sample deviation in flight time per aircraft
SELECT AIRCRAFT, STDDEV_SAMP( flying_time )
FROM flights GROUP BY aircraft;
```

SUM function

SUM is an aggregate function that evaluates the sum of the expression over a set of rows.

See Aggregates (set functions) for more information about these 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).

Examples

```
-- 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;
```

TAN function

The TAN function returns the tangent of a specified number.

The specified number is the angle, in radians, that you want the tangent for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
TAN ( number )
```

TANH function

The TANH function returns the hyperbolic tangent of a specified number.

The specified number is the angle, in radians, that you want the hyperbolic tangent for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
TANH ( number )
```

TIME function

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, LONG VARCHAR, or XML value. 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 )
```

Example

```
values time(current_timestamp)
```

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

TIMESTAMP function

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, LONG VARCHAR, or XML value. 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 ] )
```

Examples

The second column in table `records_table` contains dates (such as 1998-12-25) and the third column contains times of day (such as 17:12:30). You can return the timestamp with this statement:

```
SELECT TIMESTAMP(col2, col3) FROM records_table
```

The following clause returns the value `1998-12-25-17:12:30.0`:

```
VALUES TIMESTAMP('1998-12-25', '17.12.30');
1
--------------------------
1998-12-25 17:12:30.0
```

TRIM function

The TRIM function takes a character expression and returns that expression with leading and/or trailing pad characters removed.

Optional parameters indicate whether leading, or trailing, or both leading and trailing pad characters should be removed, and specify the pad character that is to be removed.

Syntax

```
TRIM( [ trimOperands ] trimSource )
```

The *trimSource* is a *characterExpression*.

trimOperands

```
{ trimType [ trimCharacter ] FROM | trimCharacter FROM }
```

The *trimCharacter* is a *characterExpression*.

trimType

```
{ LEADING | TRAILING | BOTH }
```

If *trimType* is not specified, it defaults to BOTH. If *trimCharacter* is not specified, it will default to the space character (' '). Otherwise the *trimCharacter* expression must evaulate to one of the following:

- A character string whose length is exactly one
- NULL

If either *trimCharacter* or *trimSource* evaluates to NULL, the result of the TRIM function is NULL. Otherwise, the result of the TRIM function is defined as follows:

- If *trimType* is LEADING, the result will be the *trimSource* value with all leading occurrences of *trimCharacter* removed.
- If *trimType* is TRAILING, the result will be the *trimSource* value with all trailing occurrences of *trimCharacter* removed.
- If *trimType* is BOTH, the result will be the *trimSource* value with all leading *and* trailing occurrences of *trimCharacter* removed.

If *trimSource*'s data type is CHAR or VARCHAR, the return type of the TRIM function will be VARCHAR. Otherwise the return type of the TRIM function will be CLOB.

Examples

```
-- returns 'derby' (no spaces)
VALUES TRIM('  derby ')

-- returns 'derby' (no spaces)
VALUES TRIM(BOTH ' ' FROM '  derby ')

-- returns 'derby ' (with a space at the end)
VALUES TRIM(LEADING ' ' FROM '  derby ')

-- returns '  derby' (with two spaces at the beginning)
VALUES TRIM(TRAILING ' ' FROM '  derby ')

-- returns NULL
VALUES TRIM(cast (null as char(1)) FROM '  derby ')

-- returns NULL
VALUES TRIM(' ' FROM cast(null as varchar(30)))

-- returns ' derb' (with a space at the beginning)
VALUES TRIM('y' FROM ' derby')
```

```
-- results in an error because trimCharacter can only be 1 character
VALUES TRIM('by' FROM ' derby')
```

UCASE or UPPER function

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

Syntax

```
UCASE ( characterExpression )
```

```
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 attribute* 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.

Example
To return the string `aSD1#w` in uppercase, use the following clause:

```
VALUES UPPER('aSD1#w')
```

The value returned is `ASD1#W`.

USER function

When used outside stored routines, the USER, CURRENT_USER, and SESSION_USER functions all return the authorization identifier of the user that created the SQL session.

See CURRENT_USER function and SESSION_USER function for details on those functions.

SESSION_USER also always returns this value when used within stored routines.

If used within a stored routine created with EXTERNAL SECURITY DEFINER, however, USER and CURRENT_USER return the authorization identifier of the user that owns the schema of the routine. This is usually the creating user, although the database owner could be the creator as well.

For information about definer's and invoker's rights, see CREATE PROCEDURE statement or CREATE FUNCTION statement.

Syntax

```
USER
```

Example

```
VALUES USER
```

VAR_POP function

VAR_POP is an aggregate function that evaluates the population variance of an expression over a set of rows.

See Aggregates (set functions) for more information about these functions.

VAR_POP is allowed only on expressions that evaluate to numeric data types.

Syntax

```
VAR_POP ( expression )
```

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 DOUBLE (it might overflow).

Formula

Population variance is calculated as follows:

$$\text{sum}(x_i^2)/n - m^2$$

where

n is the number of items in the population
m is the population average
$x_1 \ldots x_n$ are the items in the population

Example

```
-- find the variance in flight time per aircraft
SELECT AIRCRAFT, VAR_POP( flying_time )
FROM flights GROUP BY aircraft;
```

VAR_SAMP function

VAR_SAMP is an aggregate function that evaluates the sample variance of an expression over a set of rows.

See Aggregates (set functions) for more information about these functions.

VAR_SAMP is allowed only on expressions that evaluate to numeric data types.

Syntax

```
VAR_SAMP ( expression )
```

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 DOUBLE (it might overflow).

Formula

Sample variance is calculated as follows:

$$[\ \text{sum}(x_i^2) - \text{sum}(x_i)^2/n\ ]/(n-1)$$

where

n is the number of items in the population
$x_1 \ldots x_n$ are the items in the population

Example

```
-- find the sample variance in flight time per aircraft
SELECT AIRCRAFT, VAR_SAMP( flying_time )
FROM flights GROUP BY aircraft;
```

VARCHAR function

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.

Example
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'
```

XMLEXISTS operator

XMLEXISTS is an SQL/XML operator that you can use to query XML values in SQL.

The XMLEXISTS operator has two arguments, an XML query expression and a Derby XML value.

See "XML data types and operators" in the *Derby Developer's Guide* for more information.

Syntax

```
XMLEXISTS ( xqueryStringLiteral
    PASSING BY REF xmlValueExpression [ BY REF ] )
```

xqueryStringLiteral
Must be specified as a string literal. If this argument is specified as a parameter, an expression that is not a literal, or a literal that is not a string (for example an integer), Derby throws an error. The *xqueryStringLiteral* argument must also be an XPath expression. Derby does not support full XQuery, only the XPath subset. If it cannot compile or execute the query argument, Derby throws an *SQLException*. See http://www.w3.org/TR/xpath for more information on XPath expressions.

xmlValueExpression
Must be an XML data value and must constitute a well-formed SQL/XML document. The *xmlValueExpression* argument cannot be a parameter. Derby does not perform implicit parsing nor casting of XML values, so use of strings or any other data type results in an error. If the argument is a sequence that is returned by the DerbyXMLQUERY operator, the argument is accepted if it is a sequence of exactly one node that is a document node. Otherwise Derby throws an error.

BY REF
Optional keywords that describe the only value passing mechanism supported by Derby. Since BY REF is also the default passing mechanism, the XMLEXISTS

operator behaves the same whether the keywords are present or not. For more information on passing mechanisms, see the SQL/XML specification.

Operator results and combining with other operators

The result of the XMLEXISTS operator is a SQL boolean value that is based on the results from evaluating the *xqueryStringLiteral* against the *xmlValueExpression*. The XMLEXISTS operator returns:

UNKNOWN
When the *xmlValueExpression* is null.

TRUE
When the evaluation of the specified query expression against the specified *xmlValueExpression* returns a non-empty sequence of nodes or values.

FALSE
When evaluation of the specified query expression against the specified *xmlValueExpression* returns an empty sequence.

The XMLEXISTS operator does not return the actual results from the evaluation of the query. You must use the XMLQUERY operator to retrieve the actual results.

Since the result of the XMLEXISTS operator is an SQL boolean data type, you can use the XMLEXISTS operator wherever a boolean function is allowed. For example, you can use the XMLEXISTS operator as a check constraint in a table declaration or as a predicate in a WHERE clause.

Examples

In the `x_table` table, to determine if the `xcol` XML column for each row has an element called `student` with an `age` attribute equal to 20, use this statement:

```
SELECT id, XMLEXISTS('//student[@age=20]' PASSING BY REF xcol)
    FROM x_table
```

In the `x_table` table, to return the ID for every row whose `xcol` XML column is non-null and contains the element `/roster/student`, use this statement:

```
SELECT id FROM x_table WHERE XMLEXISTS('/roster/student' PASSING BY REF
 xcol)
```

You can create the `x_table` table with a check constraint that limits which XML values can be inserted into the `xcol` XML column. In this example, the constraint is that the column has at least one `student` element with an `age` attribute with a value that is less than 25. To create the table, use this statement:

```
CREATE TABLE x_table ( id INT, xcol XML
    CHECK (XMLEXISTS ('//student[@age < 25]' PASSING BY REF xcol)) )
```

XMLPARSE operator

XMLPARSE is a SQL/XML operator that you use to parse a character string expression into a Derby XML value.

You can use the result of this operator temporarily or you can store the result permanently in Derby XML columns. Whether temporary or permanent, you can use the XML value as an input to the other Derby XML operators, such as XMLEXISTS and XMLQUERY.

See "XML data types and operators" in the *Derby Developer's Guide* for more information.

Syntax

```
XMLPARSE ( DOCUMENT stringValueExpression PRESERVE WHITESPACE )
```

DOCUMENT
Required keyword that describes the type of XML input that Derby can parse. Derby can only parse string expressions that constitute well-formed XML documents, because Derby uses a parser from the *javax.xml.parsers* package to parse all string values. The parser expects the *stringValueExpression* to constitute a well-formed XML document. If the string does not constitute a well-formed document, the parser throws an error. Derby catches the error and throws the error as an *SQLException*.

stringValueExpression
Any expression that evaluates to a SQL character type, such as CHAR, VARCHAR, LONG VARCHAR, or CLOB. The *stringValueExpression* argument can also be a parameter. You must use the CAST function when you specify the parameter to indicate the type of value that is bound into the parameter. Derby must verify that the parameter is the correct data type before the value is parsed as an XML document. If a parameter is specified without the CAST function, or if the CAST is to a non-character datatype, Derby throws an error.

PRESERVE WHITESPACE
Required keywords that describe how Derby handles whitespace between consecutive XML nodes. When the PRESERVE WHITESPACE keywords are used, Derby preserves whitespace as dictated by the SQL/XML rules for preserving whitespace.

For more information on what constitutes a well-formed XML document, see the following specification: http://www.w3.org/TR/REC-xml/#sec-well-formed.

Restriction: The SQL/XML standard dictates that the argument to the XMLPARSE operator can also be a binary string. However, Derby only supports character string input for the XMLPARSE operator.

Examples
To insert a simple XML document into the `xcol` XML column in the `x_table` table, use the following statement:

```
INSERT INTO x_table VALUES (1, XMLPARSE(DOCUMENT '
        <roster>
          <student age="18">AB</student>
          <student age="23">BC</student>
          <student>NOAGE</student>
        </roster>'
      PRESERVE WHITESPACE)
    )
```

To insert a large XML document into the `xcol` XML column in the `x_table` table, from JDBC use the following statement:

```
INSERT INTO x_table VALUES(2, XMLPARSE (DOCUMENT
        CAST (? AS CLOB)
      PRESERVE WHITESPACE)
    )
```

You should bind into the statement using the *setCharacterStream()* method, or any other JDBC *setXXX* method that works for the CAST target type.

XMLQUERY operator

XMLQUERY is a SQL/XML operator that you can use to query XML values in SQL.

The XMLQUERY operator has two arguments, an XML query expression and a Derby XML value.

See "XML data types and operators" in the *Derby Developer's Guide* for more information.

Syntax

```
XMLQUERY ( xqueryStringLiteral
    PASSING BY REF xmlValueExpression
    [ RETURNING SEQUENCE [ BY REF ] ]
    EMPTY ON EMPTY
    )
```

xqueryStringLiteral
: Must be specified as a string literal. If this argument is specified as a parameter, an expression that is not a literal, or a literal that is not a string (for example an integer), Derby throws an error. The *xqueryStringLiteral* argument must also be an XPath expression. Derby does not support full XQuery, only the XPath subset. If it cannot compile or execute the query argument, Derby throws an *SQLException*. See http://www.w3.org/TR/xpath for more information on XPath expressions.

xmlValueExpression
: Must be an XML data value and must constitute a well-formed SQL/XML document. The *xmlValueExpression* argument cannot be a parameter. Derby does not perform implicit parsing nor casting of XML values, so use of strings or any other data type results in an error. If the argument is a sequence that is returned by a Derby XMLQUERY operation, the argument is accepted if it is a sequence of exactly one node that is a document node. Otherwise Derby throws an error.

BY REF
: Optional keywords that describe the only value passing mechanism supported by Derby. Since BY REF is also the default passing mechanism, the XMLQUERY operator behaves the same whether the keywords are present or not. For more information on passing mechanisms, see the SQL/XML specification.

RETURNING SEQUENCE
: Optional keywords that describe the only XML type returned by the Derby XMLQUERY operator. Since SEQUENCE is also the default return type, the XMLQUERY operator behaves the same whether the keywords are present or not. For more information on the different XML return types, see the SQL/XML specification.

EMPTY ON EMPTY
: Required keywords that describe the way in which XMLQUERY handles an empty result sequence. The XMLQUERY operator returns an empty sequence exactly as the sequence is. The XMLQUERY operator does not convert the empty sequence to a null value. When an empty result sequence is serialized, the result is an empty string. Derby does not consider an empty result sequence to be a well-formed XML document.

The result of the XMLQUERY operator is a value of type XML. The result represents a sequence of XML nodes or values. Atomic values, such as strings, can be part of the result sequence. The result of an XMLQUERY operator is not guaranteed to represent a well-formed XML document and it might not be possible to insert the result of an XMLQUERY operator into an XML column. To store the result in an XML column, the result must be a sequence with exactly one item in the sequence and the item must be a well-formed document node. The result can be viewed only in serialized form by explicitly using the XMLSERIALIZE operator.

Examples

In the `x_table` table, to search the XML column `xcol` and return the students that have an age attribute that is greater than 20, use the following statement:

```
SELECT ID,
    XMLSERIALIZE(
        XMLQUERY('//student[@age>20]' PASSING BY REF xcol EMPTY ON EMPTY)
    AS VARCHAR(50))
FROM x_table
```

The result set for this query contains a row for every row in `x_table`, regardless of whether or not the XMLQUERY operator actually returns results.

In the `x_table` table, to search the XML column `xcol` and return the ages for any students named BC, use the following statement:

```
SELECT ID,
    XMLSERIALIZE(
        XMLQUERY('string(//student[text() = "BC"]/@age)' PASSING BY REF
 xcol EMPTY ON EMPTY)
    AS VARCHAR(50))
FROM x_table
WHERE
    XMLEXISTS('//student[text() = "BC"]' PASSING BY REF xcol)
```

The result set for this query contains a row for only the rows in `x_table` that have a student whose name is BC.

XMLSERIALIZE operator

XMLSERIALIZE is a SQL/XML operator that you can use to convert an XML type to a character type. There is no other way to convert the type of a Derby XML value.

Attention: Serialization is performed based on the SQL/XML serialization rules. These rules, combined with the fact that Derby supports only a subset of the XMLSERIALIZE syntax, dictate that the results of an XMLSERIALIZE operation are not guaranteed to be in-tact copies of the original XML text. For example, assume that `[xString]` is a textual representation of a well-formed XML document. You issue the following statements:

```
INSERT INTO x_table (id, xcol)
    VALUES (3, XMLPARSE(DOCUMENT '[xString]' PRESERVE WHITESPACE));

SELECT id, XMLSERIALIZE(xcol AS VARCHAR(100))
    FROM x_table WHERE id = 3;
```

There is no guarantee that the result of the XMLSERIALIZE operator will be identical to the original `[xString]` representation. Certain transformations can occur as part of XMLSERIALIZE processing, and those transformations are defined in the SQL/XML specification. In some cases the result of XMLSERIALIZE might actually be the same as the original textual representation, but that is not guaranteed.

When an XMLSERIALIZE operator is specified as part of the top-level result set for a query, the result can be accessed from JDBC by using whatever JDBC *getXXX* methods are allowed on the *stringDataType* argument that is included in the XMLSERIALIZE syntax. If you attempt to select the contents of an XML value from a top-level result set without using the XMLSERIALIZE operator, Derby throws an error. Derby does not implicitly serialize XML values.

See "XML data types and operators" in the *Derby Developer's Guide* for more information.

Syntax

```
XMLSERIALIZE ( xmlValueExpression AS stringDataType )
```

xmlValueExpression

Can be any Derby XML value, including an XML result sequence generated by the XMLQUERY operator. The *xmlValueExpression* argument cannot be a parameter.

stringDataType

Must be a SQL character string type, such as CHAR, VARCHAR, LONG VARCHAR, or CLOB. If you specify a type that is not a valid character string type, Derby throws an error.

Examples

In the `x_table` table, to display the contents of the `xcol` XML column, use this statement:

```
SELECT ID,
    XMLSERIALIZE(xcol AS CLOB)
FROM x_table
```

To retrieve the results from JDBC, you can use the JDBC *getCharacterStream()* or *getString()* method.

To display the results of an XMLQUERY operation, use the following statement:

```
SELECT ID,
    XMLSERIALIZE(
        XMLQUERY('//student[@age>20]'
           PASSING BY REF xcol EMPTY ON EMPTY)
    AS VARCHAR(50))
FROM x_table
```

YEAR function

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 an integer between 1 and 9,999. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
YEAR ( expression )
```

Example

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 system function

The `SYSCS_UTIL.SYSCS_CHECK_TABLE` function checks the specified table, ensuring that all of its indexes are consistent with the base table.

If the table and indexes are consistent, the method returns a SMALLINT with value 1. If the table and indexes are inconsistent, the function will throw an exception.

It is recommended that you run `SYSCS_UTIL.SYSCS_CHECK_TABLE` on the tables in a database offline after you back it up. Do not discard the previous backup until you have verified the consistency of the current one. Otherwise, check consistency only if there are indications that such a check is needed (for example, if you experience hardware or operating system failure), because a consistency check can take a long time on a large

database. See "Checking database consistency" in the *Derby Server and Administration Guide* for more information.

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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

Examples

Check a single table:

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

Check all tables:

```
SELECT schemaname, tablename,
    SYSCS_UTIL.SYSCS_CHECK_TABLE(schemaname, tablename)
  FROM sys.sysschemas s, sys.systables t
  WHERE s.schemaid = t.schemaid;
```

SYSCS_UTIL.SYSCS_GET_DATABASE_NAME system function

The SYSCS_UTIL.SYSCS_GET_DATABASE_NAME function returns the name of the database of the current connection.

Syntax

```
VARCHAR(32672) SYSCS_UTIL.SYSCS_GET_DATABASE_NAME()
```

Execute privileges

By default, all users have execute privilege on this function.

SQL example

Retrieve the database name:

```
VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_NAME();
```

SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY system function

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

If the value that was set for the property is invalid, the SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY function returns the invalid value, but Derby uses the default value.

Syntax

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

An error will be returned if KEY is null.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

SQL example

Retrieve the value of the `derby.locks.deadlockTimeout` property:

```
VALUES
 SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('derby.locks.deadlockTimeout');
```

SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS system function

The `SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS` function returns a VARCHAR(32672) value representing the query execution plan and runtime 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(32672) SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()
```

Execute privileges

By default, all users have execute privileges on this function.

Example

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

SYSCS_UTIL.SYSCS_GET_USER_ACCESS system function

The `SYSCS_UTIL.SYSCS_GET_USER_ACCESS` function returns the current connection access permission for the user specified.

If no permission is explicitly set for the user, the access permission for the user is the value of the default connection mode. The default connection mode is set by using the `derby.database.defaultConnectionMode` property.

Syntax

```
SYSCS_UTIL.SYSCS_GET_USER_ACCESS (USERNAME VARCHAR(128)) RETURNS
 VARCHAR(128)
```

USERNAME
An input argument of type VARCHAR(128) that specifies the user ID in the Derby database.

The value that is returned by this function is either `fullAccess`, `readOnlyAccess`, or `noAccess`.

A return value of `noAccess` means that the connection attempt by the user will be denied because neither the `derby.database.fullAccessUsers` property nor the

derby.database.readOnlyAccessUsers property is set for the user, and the derby.database.defaultConnectionMode property is set to noAccess.

The names of the connection permissions match the existing names in use by Derby.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. See "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Derby Developer's Guide* for more information. The database owner can grant access to other users.

Example

```
VALUES SYSCS_UTIL.SYSCS_GET_USER_ACCESS ('BRUNNER')
```

SYSCS_UTIL.SYSCS_GET_XPLAIN_MODE system function

The SYSCS_UTIL.SYSCS_GET_XPLAIN_MODE function returns the current XPLAIN mode.

If the XPLAIN mode is non-zero, then statements are not actually executed, but are just compiled, and their statistics recorded in the SYSXPLAIN_* database tables. If the XPLAIN mode is zero (the default), then statements are executed normally.

See "Working with RunTimeStatistics" in *Tuning Derby* for additional information.

Syntax

```
SYSCS_UTIL.SYSCS_GET_XPLAIN_MODE() RETURNS INTEGER
```

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

Example

To determine the current value of the XPLAIN mode:

```
        values syscs_util.syscs_get_xplain_mode();
```

SYSCS_UTIL.SYSCS_GET_XPLAIN_SCHEMA system function

The SYSCS_UTIL.SYSCS_GET_XPLAIN_SCHEMA function returns the XPLAIN schema for the connection.

The default XPLAIN schema is empty, so if the XPLAIN style has not been set, the function returns the empty string. If the XPLAIN schema has been set using SYSCS_UTIL.SYSCS_SET_XPLAIN_SCHEMA, the function returns the XPLAIN schema that was set. If the XPLAIN schema is set to a non-empty value, and runtime statistics are being captured, then the runtime statistics will be stored into the SYSXPLAIN_* database tables in that schema for later analysis.

See "Working with RunTimeStatistics" in *Tuning Derby* for additional information.

Syntax

```
SYSCS_UTIL.SYSCS_GET_XPLAIN_SCHEMA () RETURNS VARCHAR
```

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

Example

To determine the current value of the XPLAIN schema:

```
values syscs_util.syscs_get_xplain_schema();
```

SYSCS_UTIL.SYSCS_PEEK_AT_IDENTITY system function

The SYSCS_UTIL.SYSCS_PEEK_AT_IDENTITY function allows users to observe the next value which will be issued for an identity column.

This function can be used in databases which have been fully upgraded to Derby Release 10.11 or higher. (See "Upgrading a database" in the *Derby Developer's Guide* for more information.) In a database which has been fully upgraded to Release 10.11 or higher, identity values are produced by internal, system-managed sequence generators. SYSCS_UTIL.SYSCS_PEEK_AT_IDENTITY allows users to observe the next identity value without having to query the SYSSEQUENCES system table.

This function has no meaning in a database which is at Release 10.10 or earlier.

Querying the SYSSEQUENCES system table does not actually return the current identity value; it only returns an upper bound on that value, that is, the end of the chunk of identity values which has been preallocated but not actually used. The SYSCS_UTIL.SYSCS_PEEK_AT_IDENTITY function shows you the very next value which will be inserted into a table. Users should never directly query the SYSSEQUENCES table, because that will cause sequence generator concurrency to slow drastically.

Syntax

```
BIGINT SYSCS_UTIL.SYSCS_PEEK_AT_IDENTITY(IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128))
```

As with all system functions, schema and object name arguments are case-sensitive.

Execute privileges

By default, all users have execute privileges on this function.

Example

```
VALUES SYSCS_UTIL.SYSCS_PEEK_AT_IDENTITY('APP', 'orders');
```

SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE system function

The SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE function allows users to observe the instantaneous current value of a sequence generator without having to query the SYSSEQUENCES system table.

Querying the SYSSEQUENCES system table does not actually return the current value; it only returns an upper bound on that value, that is, the end of the chunk of sequence values which has been preallocated but not actually used. The SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE function shows you the very next value which will be returned by a NEXT VALUE FOR clause. Users should never directly query the SYSSEQUENCES table, because that will cause sequence generator concurrency to slow drastically.

Syntax

```
BIGINT SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE(IN SCHEMANAME VARCHAR(128),
IN SEQUENCENAME VARCHAR(128))
```

As with all system functions, schema and object name arguments are case-sensitive.

Execute privileges

By default, all users have execute privileges on this function.

Example

```
VALUES SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE('APP', 'order_entry_id');
```

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_BACKUP_DATABASE system procedure

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

See "Using the backup procedures to perform an online backup" in the *Derby Server and Administration Guide* for more information on using this procedure.

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 path to a directory, where the backup should be stored. Relative paths are resolved based on the current user directory, `user.dir`, of the JVM where the database backup is occurring. Relative paths are not resolved based on the derby home directory. To avoid confusion, use the absolute path.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

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_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE system procedure

The SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE system procedure backs up the database to a specified backup directory and enables the database for log archive mode.

See "Roll-forward recovery" in the *Derby Server and Administration Guide* for more information on using this procedure.

Syntax

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE
(IN BACKUPDIR VARCHAR(32672), IN SMALLINT DELETE_ARCHIVED_LOG_FILES)
```

No result is returned from the procedure.

BACKUPDIR
: An input argument of type VARCHAR(32672) that specifies the path to a directory, where the backup should be stored. Relative paths are resolved based on the current user directory, `user.dir`, of the JVM where the database backup is occurring. Relative paths are not resolved based on the derby home directory. To avoid confusion, use the absolute path

DELETE_ARCHIVED_LOG_FILES
: If the input parameter value for the DELETE_ARCHIVED_LOG_FILES parameter is a non-zero value, online archived log files that were created before this backup will be deleted. The log files are deleted only after a successful backup.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC example

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

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE(?,
 ?)");
cs.setString(1, "c:/backupdir");
cs.setInt(2, 0);
cs.execute();
```

SQL examples

The following example backs up the database to the `c:/backupdir` directory, enables log archive mode, and does not delete any existing online archived log files:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE('c:/
backupdir', 0)
```

The following example backs up the database to the `c:/backupdir` directory and, if this backup is successful, deletes existing online archived log files:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE('c:/
backupdir', 1)
```

SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT system procedure

The SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT system procedure backs up the database to a specified backup directory and enables the database for log archive mode.

This procedure returns an error if there are any transactions in progress that have unlogged operations at the start of the backup, instead of waiting for those transactions to complete.

Syntax

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT
(IN BACKUPDIR VARCHAR(32672),
IN SMALLINT DELETE_ARCHIVED_LOG_FILES)
```

No result is returned from the procedure.

BACKUPDIR
An input argument of type VARCHAR(32672) that specifies the path to a directory, where the backup should be stored. Relative paths are resolved based on the current user directory, `user.dir`, of the JVM where the database backup is occurring. Relative paths are not resolved based on the derby home directory. To avoid confusion, use the absolute path.

DELETE_ARCHIVED_LOG_FILES
If the input parameter value for the DELETE_ARCHIVED_LOG_FILES parameter is a non-zero value, online archived log files that were created before this backup will be deleted. The log files are deleted only after a successful backup.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC example

The following example backs up the database to the `c:/backupdir` directory and enables log archive mode:

```
CallableStatement cs = conn.prepareCall
("CALL
 SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT(?,
 ?)");
cs.setString(1, "c:/backupdir");
cs.setInt(2, 0);
cs.execute();
```

SQL examples

The following example backs up the database to the `c:/backupdir` directory, enables log archive mode, and does not delete any existing online archived log files:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT('c:/
backupdir', 0)
```

The following example backs up the database to the `c:/backupdir` directory and, if this backup is successful, deletes existing online archived log files:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT('c:/
backupdir', 1)
```

SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT system procedure

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

If there are any transactions in progress with unlogged operations at the start of the backup, the SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT system procedure returns an error immediately, instead of waiting for those transactions to complete.

Syntax

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

No result is returned from the procedure.

BACKUPDIR

An input argument of type VARCHAR(32672) that specifies the path to a directory, where the backup should be stored. Relative paths are resolved based on the current user directory, `user.dir`, of the JVM where the database backup is occurring. Relative paths are not resolved based on the derby home directory. To avoid confusion, use the absolute path.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC example

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

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT(?)");
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_NOWAIT('c:/backupdir');
```

SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE system procedure

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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

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_COMPRESS_TABLE system procedure

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.

The SYSCS_UTIL.SYSCS_COMPRESS_TABLE system procedure updates statistics on all indexes as part of the index rebuilding process.

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.

Execute privileges

If authentication and SQL authorization are both enabled, all users have execute privileges on this procedure. However, in order for the procedure to run successfully on a given table, the user must be the owner of either the database or the schema in which the table resides. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

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. Only the disk space that is temporarily claimed by an external sort can be returned to the operating system prior to a COMMIT.
Tip: We recommend that you issue the `SYSCS_UTIL.SYSCS_COMPRESS_TABLE` system procedure in the 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_CREATE_USER system procedure

The `SYSCS_UTIL.SYSCS_CREATE_USER` system procedure adds a new user account to a database.

This procedure creates users for use with NATIVE authentication. For details about NATIVE authentication, see *derby.authentication.provider* and "Configuring NATIVE authentication" in the *Derby Security Guide*.

If NATIVE authentication is not already turned on when you call this procedure:

- The first user whose credentials are stored must be the database owner.
- Calling this procedure will turn on NATIVE authentication the next time the database is booted.
- Once you turn on NATIVE authentication with this procedure, it remains turned on permanently. There is no way to turn it off.

Syntax

```
SYSCS_UTIL.SYSCS_CREATE_USER(IN USERNAME VARCHAR(128),
IN PASSWORD VARCHAR(32672))
```

No result set is returned by this procedure.

USERNAME
A user name that is case-sensitive if you place the name string in double quotes. This user name is an authorization identifier. See "Authorization identifiers, user authentication, and user authorization" in the *Derby Security Guide* for more information about how these names are treated.

PASSWORD
A case-sensitive password.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring NATIVE authentication," "Configuring user authentication," and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC example

Create a user named FRED:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_CREATE_USER(?, ?)");
cs.setString(1, "fred");
cs.setString(2, "fredpassword");
cs.execute();
cs.close();
```

Create a user named FreD:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_CREATE_USER(?, ?)");
cs.setString(1, "\"FreD\"");
cs.setString(2, "fredpassword");
cs.execute();
cs.close();
```

SQL example

Create a user named FRED:

```
CALL SYSCS_UTIL.SYSCS_CREATE_USER('fred', 'fredpassword')
```

Create a user named FreD:

```
CALL SYSCS_UTIL.SYSCS_CREATE_USER('"FreD"', 'fredpassword')
```

SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE system procedure

The `SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE` system procedure disables the log archive mode and deletes any existing online archived log files if the `DELETE_ARCHIVED_LOG_FILES` input parameter is non-zero.

See "Roll-forward recovery" in the *Derby Server and Administration Guide* for more information on using this procedure.

Syntax

```
SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE(IN SMALLINT
 DELETE_ARCHIVED_LOG_FILES)
```

No result is returned from the procedure.

DELETE_ARCHIVED_LOG_FILES

If the input parameter value for the DELETE_ARCHIVED_LOG_FILES parameter is a non-zero value, then all existing online archived log files are deleted. If the parameter value is zero, then existing online archived log files are not deleted.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC example

The following example disables log archive mode for the database and deletes any existing log archive files.

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE(?)");
cs.setInt(1, 1);
cs.execute();
cs.close();
```

SQL examples

The following example disables log archive mode for the database and retains any existing log archive files:

```
CALL SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE(0);
```

The following example disables log archive mode for the database and deletes any existing log archive files:

```
CALL SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE(1);
```

SYSCS_UTIL.SYSCS_DROP_STATISTICS system procedure

The SYSCS_UTIL.SYSCS_DROP_STATISTICS system procedure drops all existing cardinality statistics for the index that you specify or for all of the indexes on a table.

You may want to drop the statistics if you are no longer using them or if they become incorrect for some reason. You can call the SYSCS_UTIL.SYSCS_UPDATE_STATISTICS or SYSCS_UTIL.SYSCS_COMPRESS_TABLE system procedure to recreate them, or you can wait for automatic statistics generation to begin again.

For more information on cardinality statistics, see "Working with cardinality statistics" in the *Tuning Derby* guide.

Syntax

```
SYSCS_UTIL.SYSCS_DROP_STATISTICS(IN SCHEMANAME VARCHAR(128),
                                 IN TABLENAME VARCHAR(128),
                                 IN INDEXNAME VARCHAR(128))
```

Note: You can specify `null` for the INDEXNAME to drop all existing statistics.

Execute privileges

If authentication and SQL authorization are both enabled, all users have execute privileges on this procedure. However, in order for the procedure to run successfully on a given table, the user must be the owner of either the database or the schema in which the table resides. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

Examples

In the following example, the system procedure drops statistics for the index PAY_DESC on the SAMP.EMPLOYEE table:

```
CALL SYSCS_UTIL.SYSCS_DROP_STATISTICS('SAMP','EMPLOYEE','PAY_DESC');
```

In the following example, `null` is specified instead of an index name. For all of the indexes on the EMPLOYEE table in the SAMP schema, the existing statistics are dropped.

```
CALL SYSCS_UTIL.SYSCS_DROP_STATISTICS('SAMP', 'EMPLOYEE', null);
```

SYSCS_UTIL.SYSCS_DROP_USER system procedure

The SYSCS_UTIL.SYSCS_DROP_USER system procedure removes a user account from a database.

This procedure is used in conjunction with NATIVE authentication. For details about NATIVE authentication, see *derby.authentication.provider* and "Configuring NATIVE authentication" in the *Derby Security Guide*.

You are not allowed to remove the user account of the database owner.

If you use this procedure to remove a user account, the schemas and data objects owned by the user remain in the database and can be accessed only by the database owner or by other users who have been granted access to them. If the user is created again, the user regains access to the schemas and data objects.

Syntax

```
SYSCS_UTIL.SYSCS_DROP_USER(IN USERNAME VARCHAR(128))
```

No result set is returned by this procedure.

USERNAME

A user name that is case-sensitive if you place the name string in double quotes. This user name is an authorization identifier. See "Authorization identifiers, user authentication, and user authorization" in the *Derby Security Guide* for more information about how these names are treated. If the user name is that of the database owner, an error is raised.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring NATIVE authentication," "Configuring user authentication," and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC example

Drop a user named FRED:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_DROP_USER('fred')");
cs.execute();
cs.close();
```

SQL example

Drop a user named FreD:

```
CALL SYSCS_UTIL.SYSCS_DROP_USER('"FreD"')
```

SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE system procedure

The SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE stored procedure removes as many compiled statements (plans) as possible from the database-wide statement cache.

The procedure does not remove statements related to currently executing queries or to activations that are about to be garbage collected, so the cache is not guaranteed to be completely empty after a call to this procedure.

Syntax

```
SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE()
```

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC Example

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

SQL Example

```
CALL SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE();
```

SYSCS_UTIL.SYSCS_EXPORT_QUERY system procedure

The SYSCS_UTIL.SYSCS_EXPORT_QUERY system procedure exports the results of a SELECT statement to an operating system file.

For security concerns, and to avoid accidental file damage, this EXPORT procedure does not export data into an existing file. You must specify a filename in the EXPORT procedure that does not exist. When you run the procedure the file is created and the data is exported into the new file.

The data is exported using a delimited file format.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

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
: Specifies the name of a new file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export procedure returns an error. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have SELECT privileges on the table.

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);
```

SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE system procedure

Use the `SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE` system procedure to export the result of a SELECT statement to a main export file, and place the LOB data into a separate export file. A reference to the location of the LOB data is placed in the LOB column in the main export file.

For security concerns, and to avoid accidental file damage, this EXPORT procedure does not export data into an existing file. You must specify a filename in the EXPORT procedure that does not exist. When you run the procedure the file is created and the data is exported into the new file.

The data is exported using a delimited file format.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

Syntax

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

```
    IN LOBSFILENAME VARCHAR(32672)
    )
```

When you run this procedure, the column data is written to the main export file in a delimited data file format.

SELECTSTATEMENT
Specifies the SELECT statement query that returns the data to be exported. Specifying a NULL value will result in an error. The SELECTSTATEMENT parameter takes an input argument that is a VARCHAR (32672) data type.

FILENAME
Specifies the name of a new file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export procedure returns an error. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter must be a CHAR (1) data type.

CHARACTERDELIMITER
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CODESET
Specifies the code set of the data in the export file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the database code page to the specified code page before writing to the file. You can specify a NULL value to write the data in the same code page as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR (128) data type.

LOBSFILENAME
Specifies the file that the large object data is exported to. If the path is omitted, the lob file is created in the same directory as the main export file. If you specify the name of an existing file, the export utility overwrites the contents of the file. The data is not appended to the file. If you are using the Network Server, the file should be in a server-side location. Specifying a NULL value results in an error. The LOBSFILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have SELECT privileges on the table.

Example exporting data from a query using a separate export file for the LOB data

The following example shows how to export employee data in department 20 from the STAFF table in a sample database to the main file `staff.del` and the lob data to the file `pictures.dat`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE(
    'SELECT * FROM STAFF WHERE dept=20',
    'c:\data\staff.del', ',' ,'"',
    'UTF-8','c:\data\pictures.dat');
```

SYSCS_UTIL.SYSCS_EXPORT_TABLE system procedure

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

For security concerns, and to avoid accidental file damage, this EXPORT procedure does not export data into an existing file. You must specify a filename in the EXPORT procedure that does not exist. When you run the procedure the file is created and the data is exported into the new file.

The data is exported using a delimited file format.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

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
Specifies the name of a new file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export procedure returns an error. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

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 uppercase 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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have SELECT privileges on the table.

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);
```

SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE system procedure

Use the `SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE` system procedure to export all the data from a table, and place the LOB data into a separate export file. A reference to the location of the LOB data is placed in the LOB column in the main export file.

For security concerns, and to avoid accidental file damage, this EXPORT procedure does not export data into an existing file. You must specify a filename in the EXPORT procedure that does not exist. When you run the procedure the file is created and the data is exported into the new file.

The data is exported using a delimited file format.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

Syntax

```
SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE (
    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 LOBSFILENAME VARCHAR(32672)
    )
```

When you run this procedure, the column data is written to the main export file in a delimited data file format.

SCHEMANAME
Specifies the schema of the table. You can specify a NULL value to use the default schema name. The SCHEMANAME parameter takes an input argument that is a VARCHAR (128) data type.

TABLENAME
Specifies the table name of the table or view from which the data is to be exported. This table cannot be a system table or a declared temporary table. The string must exactly match the case of the table name. Specifying a NULL value results in an error. The TABLENAME parameter takes an input argument that is a VARCHAR (128) data type.

FILENAME
Specifies the name of a new file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export procedure returns an error. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter must be a CHAR (1) data type.

CHARACTERDELIMITER
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CODESET
Specifies the code set of the data in the export file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the database code page to the specified code page before writing to the file. You can specify a NULL value to write the data in the same code page as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR (128) data type.

LOBSFILENAME
Specifies the file that the large object data is exported to. If the path is omitted, the lob file is created in the same directory as the main export file. If you specify the name of an existing file, the export utility overwrites the contents of the file. The data is not appended to the file. If you are using the Network Server, the file should be in a server-side location. Specifying a NULL value results in an error. The LOBSFILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the export procedure using all uppercase characters. If you created a schema or table 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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have SELECT privileges on the table.

Example exporting all data from a table, using a separate export file for the LOB data

The following example shows how to export data from the STAFF table in a sample database to the main file `staff.del` and the LOB export file `pictures.dat`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE(
    'APP', 'STAFF', 'c:\data\staff.del', ',' ,'"',
```

```
    'UTF-8', 'c:\data\pictures.dat');
```

SYSCS_UTIL.SYSCS_FREEZE_DATABASE system procedure

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

See "Using operating system commands with the freeze and unfreeze system procedures to perform an online backup" in the *Derby Server and Administration Guide* for more information on using this procedure.

Syntax

```
SYSCS_UTIL.SYSCS_FREEZE_DATABASE()
```

No result set is returned by this procedure.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

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_IMPORT_DATA system procedure

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.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

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(32672) 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(32672) that specifies the indexes (numbered from 1 and separated by commas) of the input data fields to be imported. This argument cannot identify column names as in `SYSCS_UTIL.SYSCS_IMPORT_DATA_BULK procedure`. 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 uppercase 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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have INSERT privileges on the table.

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)
```

SYSCS_UTIL.SYSCS_IMPORT_DATA_BULK system procedure

The SYSCS_UTIL.SYSCS_IMPORT_DATA_BULK system procedure supports all functionalities of SYSCS_UTIL.SYSCS_IMPORT_DATA with the additional feature to skip column headers in the input file and recognize columns in the input file by name that are parsed to the COLUMNINDEXES argument.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_DATA_BULK (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,
IN SKIP 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(32672) 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(32672) that specifies the indexes (numbered from 1 and separated by commas) and column names (double quoted, case sensitive and seperated by commas) of the input data fields to be imported. The input file columns can only be referenced by the column names if the SKIP argument is greater than 0. 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

An 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.

SKIP

An input argument of type SMALLINT. SKIP number of header lines will be ignored and rest of lines in the input file will be imported to the table

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 uppercase 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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have INSERT privileges on the table.

Example

The following examples import data fields from a delimited data file called `petlist.del` into the PET table ignoring SKIP number of header lines and importing rest of the input file. This examples also show parsing column name to the COLUMNINDEXES argument.

Example 1

This example shows one header line in the input file. Data contained in the input file petlist.del is given below.

```
Pet Name,Kind of Animal,Age
Rover,Dog,4
Spot,cat,2
Squawky,Parrot,37
```

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA_BULK
    (NULL, 'PET', NULL, '\"Pet Name\",2,3', 'petlist.del', NULL, NULL,
 NULL, 0, 1)
```

This procedure identifies the column by name as given in the example. When data is imported from the input file, first line in the input file is ignored. That is the number of lines given in the SKIP argument. Rest of the lines in the input file are imported. This means that first line

```
Pet Name,Kind of Animal,Age
```

is ignored and next three lines,

```
Rover,Dog,4
Spot,cat,2
Squawky,Parrot,37
```

are imported.

Example 2

This example shows three header lines in the input file. Data contained in the input file petlist.del is given below.

```
Pet,Kind,Age
Name,of,
,Animal,
Rover,Dog,4
Spot,cat,2
Squawky,Parrot,37
```

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA_BULK
(NULL, 'PET', NULL, '1,\"Kind of Animal\",3', 'petlist.del', NULL,
 NULL, NULL, 0, 3)
```

When data is imported from the input file, first three lines in the input file are ignored. That is the number of lines given in the SKIP argument. Rest of the lines in the input file are imported. This means that first three lines

```
Pet,Kind,Age
Name,of,
,Animal,
```

are ignored and next three lines,

```
Rover,Dog,4
Spot,cat,2
Squawky,Parrot,37
```

are imported.

SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE system procedure

Use the SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE system procedure to import data to a subset of columns in a table, where the LOB data is stored in a separate file. The main import file contains all of the other data and a reference to the location of the LOB data.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE (
   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)
   )
```

The import utility looks in the main import file for a reference to the location of the LOB data.

SCHEMANAME
Specifies the schema of the table. You can specify a NULL value to use the default schema name. The SCHEMANAME parameter takes an input argument that is a VARCHAR(128) data type.

TABLENAME
Specifies the name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. The string must exactly match case of the table name. Specifying a NULL value results in an error. The TABLENAME parameter takes an input argument that is a VARCHAR(128) data type.

INSERTCOLUMNS
Specifies the comma separated column names of the table into which the data will be imported. You can specify a NULL value to import into all columns of the table. The INSERTCOLUMNS parameter takes an input argument that is a VARCHAR(32672) data type.

COLUMNINDEXES
Specifies the comma separated column indexes (numbered from one) of the input data fields that will be imported. You can specify a NULL value to use all input data fields in the file. The COLUMNINDEXES parameter takes an input argument that is a VARCHAR(32672) data type. This argument cannot identify column names as in `SYSCS_UTIL.SYSCS_IMPORT_DATA_BULK procedure`.

FILENAME
Specifies the name of the file that contains the data to be imported. If the path is omitted, the current working directory is used. The specified location of the file should refer to the server side location if using the Network Server. Specifying a NULL value results in an error. The fileName parameter takes an input argument that is a VARCHAR(32672) data type.

COLUMNDELIMITER
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter takes an input argument that is a CHAR(1) data type.

CHARACTERDELIMITER
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR(1) data type.

CODESET
Specifies the code set of the data in the input file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the specified code set to the database code set (UTF-8). You can specify a NULL value to interpret the data file in the same code set as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR(128) data type.

REPLACE
A non-zero value for the replace parameter will import in REPLACE mode, while a zero value will import in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the table and inserts the imported data. The table definition and the index definitions are not changed. You can only import with REPLACE mode if the table already exists. INSERT mode adds the imported data to the table without changing the existing table data. Specifying a NULL value results in

an error. The REPLACE parameter takes an input argument that is a SMALLINT data type.

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 uppercase 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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have INSERT privileges on the table.

Usage

This procedure will read the LOB data using the reference that is stored in the main import file. The format of the reference to the LOB stored in the main import file must be `lobsFileName.Offset.length/`.

- `Offset` is position in the external file in bytes
- `length` is the size of the LOB column data in bytes

Example importing data into specific columns, using a separate import file for the LOB data

The following example shows how to import data into several columns of the STAFF table. The STAFF table includes a LOB column in a sample database. The import file `staff.del` is a delimited data file. The `staff.del` file contains references that point to a separate file which contains the LOB data. The data in the import file is formatted using double quotation marks (") as the string delimiter and a comma (,) as the column delimiter. The data will be appended to the existing data in the STAFF table.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE
    (null, 'STAFF', 'NAME,DEPT,SALARY,PICTURE', '2,3,4,6',
    'c:\data\staff.del', ',','"','UTF-8',  0);
```

SYSCS_UTIL.SYSCS_IMPORT_TABLE system procedure

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.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

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 uppercase 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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have INSERT privileges on the table.

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);
```

SYSCS_UTIL.SYSCS_IMPORT_TABLE_BULK system procedure

The SYSCS_UTIL.SYSCS_IMPORT_TABLE_BULK system procedure supports all functionalities of SYSCS_UTIL.SYSCS_IMPORT_TABLE with the additional feature to skip column headers in the input file.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_TABLE_BULK (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
IN SKIP 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
An 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.

SKIP
An input argument of type SMALLINT. SKIP number of header lines will be ignored and rest of lines in the input file will be imported to the table

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 uppercase 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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have INSERT privileges on the table.

Example

The following example imports data into the `PET` table from a delimited data file called `petlist.del` Data contained in the input file petlist.del is given below.

```
Pet Name,Kind of Animal,Age
Rover,Dog,4
Spot,cat,2
Squawky,Parrot,37
```

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE_BULK
(null, 'PET', 'c:/output/petlist.del', null, null, null, 0, 1);
```

When data is imported from the input file, first line in the input file is ignored. That is the number of lines given in the SKIP argument. Rest of the lines in the input file is imported. This means that first line

```
Pet Name,Kind of Animal,Age
```

is ignored and other three lines,

```
Rover,Dog,4
Spot,cat,2
Squawky,Parrot,37
```

are imported.

SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE system procedure

Use the `SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE` system procedure to import data to a table, where the LOB data is stored in a separate file. The main import file contains all of the other data and a reference to the location of the LOB data.

Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure.

For more information on using this procedure, see the section "Importing and exporting data" in the *Derby Server and Administration Guide*.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE (
   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)
   )
```

The import utility looks in the main import file for a reference to the location of the LOB data.

SCHEMANAME
Specifies the schema of the table. You can specify a NULL value to use the default schema name. The SCHEMANAME parameter takes an input argument that is a VARCHAR (128) data type.

TABLENAME
Specifies the name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. The string must exactly match case of the table name. Specifying a NULL value results in an error. The TABLENAME parameter takes an input argument that is a VARCHAR (128) data type.

FILENAME
Specifies the name of the file that contains the data to be imported. If the path is omitted, the current working directory is used. The specified location of the file should refer to the server side location if using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CHARACTERDELIMITER
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CODESET
Specifies the code set of the data in the input file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the specified code set to the database code set (UTF-8). You can specify a NULL value to interpret the data file in the same code set as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR (128) data type.

REPLACE
A non-zero value for the replace parameter will import in REPLACE mode, while a zero value will import in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the table and inserts the imported data. The table definition and the index definitions are not changed. You can only import with REPLACE mode if the table already exists. INSERT mode adds the imported data to the table without changing the existing table data. Specifying a NULL value results in an error. The REPLACE parameter takes an input argument that is a SMALLINT data type.

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 uppercase 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.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users. The user must also have INSERT privileges on the table.

Usage

This procedure will read the LOB data using the reference that is stored in the main import file. If you are importing from a non-Derby source, the format of the reference to the LOB stored in the main import file must be `lobsFileName.Offset.length/`.

- `Offset` is position in the external file in bytes
- `length` is the size of the LOB column data in bytes

Example importing data from a main import file that contains references which point to a separate file that contains LOB data

The following example shows how to import data into the *STAFF* table in a sample database from a delimited data file `staff.del`. This example defines a comma as the column delimiter. The data will be appended to the existing data in the table.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE(
    'APP','STAFF','c:\data\staff.del',',','"','UTF-8',0);
```

SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE system procedure

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 have 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.

Execute privileges

If authentication and SQL authorization are both enabled, all users have execute privileges on this procedure. However, in order for the procedure to run successfully on a given table, the user must be the owner of either the database or the schema in which the table resides. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

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_INPLACE_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_INPLACE_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();
```

Tip: We recommend that you issue the SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE system 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_INVALIDATE_STORED_STATEMENTS system procedure

The SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS system procedure invalidates all stored prepared statements (that is, all statements in the SYSSTATEMENTS system table).

The next time one of the invalid stored prepared statements in the SYSSTATEMENTS system table is executed, it will be recompiled, and a new plan will be generated for it.

Run SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS whenever you think that your metadata queries or triggers are misbehaving -- for example, if they throw `java.lang.NoSuchMethodError` or `java.lang.NoSuchMethodException` on execution. Derby stores plans for triggers and metadata queries in the database. These should be invalidated automatically on upgrade and at other necessary times. Should you encounter an instance where they are not, you have found a bug that you should report, but one that you can likely work around by running SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS.

Syntax

```
SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS()
```

No result is returned by this procedure.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC example

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

SQL Example

```
CALL SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS();
```

SYSCS_UTIL.SYSCS_MODIFY_PASSWORD system procedure

The SYSCS_UTIL.SYSCS_MODIFY_PASSWORD system procedure is called by a user to change that user's own password.

This procedure is used in conjunction with NATIVE authentication. For details about NATIVE authentication, see *derby.authentication.provider* and "Configuring NATIVE authentication" in the *Derby Security Guide*.

The *derby.authentication.native.passwordLifetimeMillis* property sets the password expiration time, and the *derby.authentication.native.passwordLifetimeThreshold* property sets the time when a user is warned that the password will expire.

Syntax

```
SYSCS_UTIL.SYSCS_MODIFY_PASSWORD(IN PASSWORD VARCHAR(32672))
```

No result set is returned by this procedure.

PASSWORD
A case-sensitive password.

Execute privileges

Any user can execute this procedure.

JDBC example

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_MODIFY_PASSWORD('shhhhh!')");
cs.execute();
cs.close();
```

SQL example

```
CALL SYSCS_UTIL.SYSCS_MODIFY_PASSWORD('shhhhh!')
```

SYSCS_UTIL.SYSCS_REGISTER_TOOL system procedure

The `SYSCS_UTIL.SYSCS_REGISTER_TOOL` system procedure loads and unloads optional tools packages.

Syntax

```
SYSCS_UTIL.SYSCS_REGISTER_TOOL(IN TOOLNAME VARCHAR(128),
                               IN REGISTER BOOLEAN,
                               IN OPTIONALARGS VARCHAR(128) ... )
```

No result set is returned by this procedure.

TOOLNAME
The name of the optional tool. Must be one of the following case-sensitive names: `'databaseMetaData'`, `'foreignViews'`, `'luceneSupport'`, `'simpleJson'`, or `'rawDBReader'`.

REGISTER
A value of true tells Derby to load the tool. A value of false tells Derby to unload the tool.

OPTIONALARGS
Optional case-sensitive string arguments specific to each tool.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

Usage

The following optional tools are supported:

- `databaseMetaData`

This optional tool creates functions and table functions to wrap the methods in *java.sql.DatabaseMetaData*, allowing you to use *DatabaseMetaData* methods in queries. For example, you can join and filter the *ResultSet*s returned by *DatabaseMetaData* methods. This tool does not require any optional arguments. To create the metadata functions and table functions, do the following:

```
call syscs_util.syscs_register_tool( 'databaseMetaData', true )
```

To drop the functions and table functions, do the following:

```
call syscs_util.syscs_register_tool( 'databaseMetaData', false )
```

- `foreignViews`

  This optional tool creates schemas, table functions, and convenience views for all user tables in a foreign database. The table functions and views are useful for bulk-importing foreign data into Derby. This tool takes two additional arguments:

  CONNECTION_URL
  This is a connection URL string suitable for creating a connection to the foreign database by calling *DriverManager.getConnection()*.

  SCHEMA_PREFIX
  This is an optional string prefixed to all of the schema names which the tool creates. This argument may be omitted. If it is omitted, then the tool will create schemas which have the same names as the schemas in the foreign database.

  To create views on the foreign data, do the following:

```
call syscs_util.syscs_register_tool( 'foreignViews', true,
    'foreignDatabaseURL', 'XYZ_' )
```

  To drop the views on the foreign data, do the following:

```
call syscs_util.syscs_register_tool( 'foreignViews', false,
    'foreignDatabaseURL', 'XYZ_' )
```

See the *Derby Tools and Utilities Guide* for more information on how to use these tools. Before you run an optional tool, make sure that your classpath contains the Derby jar files, including *derbytools.jar*.

SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY system procedure

The `SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY` system procedure reloads the security policy, allowing you to fine-tune your Java security on the fly.

For more information on security policies, see "Configuring Java security" in the *Derby Security Guide*.

Syntax

```
SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY()
```

No result set is returned by this procedure.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

Example

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

SYSCS_UTIL.SYSCS_RESET_PASSWORD system procedure

The SYSCS_UTIL.SYSCS_RESET_PASSWORD system procedure resets a password that has expired or has been forgotten.

This procedure is used in conjunction with NATIVE authentication. For details about NATIVE authentication, see *derby.authentication.provider* and "Configuring NATIVE authentication" in the *Derby Security Guide*.

Syntax

```
SYSCS_UTIL.SYSCS_RESET_PASSWORD(IN USERNAME VARCHAR(128),
IN PASSWORD VARCHAR(32672))
```

No result set is returned by this procedure.

USERNAME
A user name that is case-sensitive if you place the name string in double quotes. This user name is an authorization identifier. See "Authorization identifiers, user authentication, and user authorization" in the *Derby Security Guide* for more information about how these names are treated.

PASSWORD
A case-sensitive password.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring NATIVE authentication," "Configuring user authentication," and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

JDBC example

Reset the password of a user named FRED:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_RESET_PASSWORD(?, ?)");
cs.setString(1, "fred");
cs.setString(2, "temppassword");
cs.execute();
cs.close();
```

Reset the password of a user named FreD:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_RESET_PASSWORD(?, ?)");
cs.setString(1, "\"FreD\"");
cs.setString(2, "temppassword");
cs.execute();
cs.close();
```

SQL example

Reset the password of a user named FRED:

```
CALL SYSCS_UTIL.SYSCS_RESET_PASSWORD('fred', 'temppassword')
```

Reset the password of a user named FreD:

```
CALL SYSCS_UTIL.SYSCS_RESET_PASSWORD('"FreD"', 'temppassword')
```

SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure

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.

For information about properties, see Derby property reference.

Syntax

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

This procedure does not return any results.

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.

If VALUE is an invalid value for the property, Derby uses the default value of the property, although SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY sets the invalid value.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

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_SET_RUNTIMESTATISTICS system procedure

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)
```

Execute privileges

By default, all users have execute privileges on this procedure.

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 system procedure

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)
```

Execute privileges

By default, all users have execute privileges on this procedure.

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_USER_ACCESS system procedure

The SYSCS_UTIL.SYSCS_SET_USER_ACCESS system procedure sets the connection access permission for the user specified.

Syntax

```
SYSCS_UTIL.SYSCS_SET_USER_ACCESS (USERNAME VARCHAR(128),
CONNECTION_PERMISSION VARCHAR(128))
```

USERNAME

An input argument of type VARCHAR(128) that specifies the user ID in the Derby database.

CONNECTION_PERMISSION

Valid values for CONNECTION_PERMISSION are:

FULLACCESS
Adds the user to the list of users with full access to the database. The value for the database property is `derby.database.fullAccessUsers`.

READONLYACCESS
Adds the user to the list of users with read-only access to the database. The value for the database property is `derby.database.readOnlyAccessUsers`.

null
Removes the user from the list of permissions, reverting the user to the default permission. You must specify `null` without the quotation marks.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

Example

```
CALL SYSCS_UTIL.SYSCS_SET_USER_ACCESS ('BRUNNER', 'READONLYACCESS')
```

To remove the user from the list of permissions, you specify the `null` value without the quotation marks. For example:

```
CALL SYSCS_UTIL.SYSCS_SET_USER_ACCESS ('ISABEL', null)
```

SYSCS_UTIL.SYSCS_SET_XPLAIN_MODE system procedure

When runtime statistics are being captured, you can control the mode of processing by using the `SYSCS_UTIL.SYSCS_SET_XPLAIN_MODE` procedure.

When the XPLAIN mode is set to 1, statements are compiled and optimized, but not executed; when the XPLAIN mode is set to 0 (the default), statements are compiled, optimized, and executed normally.

The XPLAIN mode only matters when XPLAIN style has been enabled. See the `SYSCS_UTIL.SYSCS_SET_XPLAIN_SCHEMA` system procedure for more information.

See "Working with RunTimeStatistics" in *Tuning Derby* for additional information.

Syntax

```
SYSCS_UTIL.SYSCS_SET_XPLAIN_MODE(IN SMALLINT NOEXECUTE)
```

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

Example

To let Derby explain a statement without executing it:

```
call syscs_util.syscs_set_runtimestatistics(1);
call syscs_util.syscs_set_xplain_schema('STATS');
call syscs_util.syscs_set_xplain_mode(1);

select country from countries;
```

```
        call syscs_util.syscs_set_runtimestatistics(0);
        call syscs_util.syscs_set_xplain_schema('');
        call syscs_util.syscs_set_xplain_mode(0);
```

SYSCS_UTIL.SYSCS_SET_XPLAIN_SCHEMA system procedure

The SYSCS_UTIL.SYSCS_SET_XPLAIN_SCHEMA system procedure allows you to request XPLAIN style processing of runtime statistics.

When XPLAIN style is used, the runtime statistics are written to the SYSXPLAIN_* database tables, so that you can analyze the statistics by running queries against the tables.

See "Working with RunTimeStatistics" in *Tuning Derby* for additional information.

Turn XPLAIN style on by calling this procedure with a non-empty argument. Turn XPLAIN style off by calling the procedure with an empty argument.

The argument that you provide must be a legal schema name, and you should use this argument to indicate the schema in which runtime statistics should be captured. If the schema that you specify does not already exist, it will be automatically created. If the XPLAIN tables do not already exist in this schema, they will be automatically created. Runtime statistics information about statements executed in this session will then be captured into these tables, until runtime statistics capturing is halted either by calling SYSCS_UTIL.SYSCS_SET_XPLAIN_SCHEMA with an empty argument or by calling SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(0).

Syntax

```
SYSCS_UTIL.SYSCS_SET_XPLAIN_SCHEMA(IN VARCHAR(128) SCHEMA_NAME)
```

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

Example

To cause Derby to record statistics about statement execution in the SYSXPLAIN_* database tables in the schema named 'MY_STATS':

```
        call syscs_util.syscs_set_runtimestatistics(1);
        call syscs_util.syscs_set_xplain_schema('MY_STATS');

        select country from countries;

        call syscs_util.syscs_set_runtimestatistics(0);
        call syscs_util.syscs_set_xplain_schema('');
```

SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE system procedure

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

See "Using operating system commands with the freeze and unfreeze system procedures to perform an online backup" in the *Derby Server and Administration Guide* for more information on using this procedure.

Syntax

```
SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()
```

No result set is returned by this procedure.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

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_UPDATE_STATISTICS system procedure

The SYSCS_UTIL.SYSCS_UPDATE_STATISTICS system procedure updates the cardinality statistics, or creates the statistics if they do not exist, for the index that you specify or for all of the indexes on a table.

Derby uses cardinality statistics to determine the optimal query plan during the compilation of a query. If the statistics are missing, Derby might use a query plan which is not the most efficient plan.

Once statistics have been created, they should be maintained. It is a good idea to call the SYSCS_UTIL.SYSCS_UPDATE_STATISTICS procedure when the number of distinct values in an index is likely to have changed significantly. To drop all existing statistics and start again from scratch, call the SYSCS_UTIL.SYSCS_DROP_STATISTICS system procedure.

For more information on cardinality statistics, see "Working with cardinality statistics" in *Tuning Derby*.

Syntax

```
SYSCS_UTIL.SYSCS_UPDATE_STATISTICS(IN SCHEMANAME VARCHAR(128),
                                   IN TABLENAME VARCHAR(128),
                                   IN INDEXNAME VARCHAR(128))
```

Note: You can specify `null` for the INDEXNAME to update any existing statistics and create statistics for those statistics that are missing.

Execute privileges

If authentication and SQL authorization are both enabled, all users have execute privileges on this procedure. However, in order for the procedure to run successfully on a given table, the user must be the owner of either the database or the schema in which the table resides. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

Examples

In the following example, the system procedure updates statistics for the index PAY_DESC on the SAMP.EMPLOYEE table:

```
CALL SYSCS_UTIL.SYSCS_UPDATE_STATISTICS('SAMP','EMPLOYEE','PAY_DESC');
```

In the following example, `null` is specified instead of an index name. For all of the indexes, the existing statistics are updated and statistics are created for any missing statistics on the EMPLOYEE table in the SAMP schema.

```
CALL SYSCS_UTIL.SYSCS_UPDATE_STATISTICS('SAMP', 'EMPLOYEE', null);
```

System procedures for storing jar files in a database

`SQLJ.INSTALL_JAR`, `SQLJ.REMOVE_JAR`, and `SQLJ.REPLACE_JAR` are a set of procedures in the SQLJ schema that allow you to store jar files in a database.

Your jar file has two names:

- A *physical name* (the name you gave it when you created it)
- A *Derby name* (the Derby identifier you give it when you load it into a particular schema). The Derby name, an *SQLIdentifier*, can be delimited and must be unique within a schema.

A single schema can store more than one jar file.

For more information on when and how to use these procedures, see "Loading classes from a database" in the *Derby Developer's Guide*.

SQLJ.INSTALL_JAR system procedure

The `SQLJ.INSTALL_JAR` system procedure stores a jar file in a database.

Syntax

```
SQLJ.INSTALL_JAR(IN JAR_FILE_PATH_OR_URL VARCHAR(32672),
                 IN QUALIFIED_JAR_NAME VARCHAR(32672),
                 IN DEPLOY INTEGER)
```

JAR_FILE_PATH_OR_URL
The path or URL of the jar file to add. A path includes both the directory and the file name (unless the file is in the current directory, in which case the directory is optional). Two examples:

```
d:/todays_build/tours.jar
```

```
http://www.example.com/tours.jar
```

QUALIFIED_JAR_NAME
The Derby name of the jar file, qualified by the schema name. Two examples:

```
MYSCHEMA.Sample1
```

```
 -- a delimited identifier
MYSCHEMA."Sample2"
```

DEPLOY
If set to 1, indicates the existence of an SQLJ deployment descriptor file. Derby ignores this argument, so it is normally set to 0.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

The database owner can grant access to other users. Since this procedure can be used to install arbitrary code (possibly from across the network) that runs in the same Java

Virtual Machine as the Derby database engine, the execute privilege should be granted only to trusted users.

SQL examples

```
-- SQL statement
-- install jar from current directory
CALL SQLJ.INSTALL_JAR('tours.jar', 'APP.Sample1', 0)

-- SQL statement
-- install jar using full path
CALL SQLJ.INSTALL_JAR('c:\myjarfiles\tours.jar', 'APP.Sample1', 0)

-- SQL statement
-- install jar from remote location
CALL SQLJ.INSTALL_JAR('http://www.example.com/tours.jar', 'APP.Sample2',
 0)

-- SQL statement
-- install jar using a quoted identifier for the
-- Derby jar name
CALL SQLJ.INSTALL_JAR('tours.jar', 'APP."Sample3"', 0)
```

SQLJ.REMOVE_JAR system procedure

The SQLJ.REMOVE_JAR system procedure removes a jar file from a database.

Syntax

```
SQLJ.REMOVE_JAR(IN QUALIFIED_JAR_NAME VARCHAR(32672),
                IN UNDEPLOY INTEGER)
```

QUALIFIED_JAR_NAME

The Derby name of the jar file, qualified by the schema name. Two examples:

```
MYSCHEMA.Sample1
```

```
 -- a delimited identifier.
MYSCHEMA."Sample2"
```

UNDEPLOY

If set to 1, indicates the existence of an SQLJ deployment descriptor file. Derby ignores this argument, so it is normally set to 0.

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information. The database owner can grant access to other users.

SQL example

```
-- SQL statement
CALL SQLJ.REMOVE_JAR('APP.Sample1', 0)
```

SQLJ.REPLACE_JAR system procedure

The SQLJ.REPLACE_JAR system procedure replaces a jar file in a database.

Syntax

```
SQLJ.REPLACE_JAR(IN JAR_FILE_PATH_OR_URL VARCHAR(32672),
                 IN QUALIFIED_JAR_NAME VARCHAR(32672))
```

JAR_FILE_PATH_OR_URL

The path or URL of the jar file to use as a replacement. A path includes both the directory and the file name (unless the file is in the current directory, in which case the directory is optional). For example:

```
d:/todays_build/tours.jar
```

QUALIFIED_JAR_NAME
The Derby name of the jar file, qualified by the schema name. Two examples:

```
MYSCHEMA.Sample1
```

```
 -- a delimited identifier.
MYSCHEMA."Sample2"
```

Execute privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this procedure by default. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

The database owner can grant access to other users. Since this procedure can be used to install arbitrary code (possibly from across the network) that runs in the same Java Virtual Machine as the Derby database engine, the execute privilege should be granted only to trusted users.

SQL example

```
-- SQL statement
CALL sqlj.replace_jar('c:\myjarfiles\newtours.jar', 'APP.Sample1')

-- SQL statement
-- replace jar from remote location
CALL SQLJ.REPLACE_JAR('http://www.example.com/tours.jar', 'APP.Sample2')
```

SYSCS_DIAG diagnostic tables and functions

Derby provides a set of system table expressions which you can use to obtain diagnostic information about the state of the database and about the database sessions.

There are two types of diagnostic table expressions in Derby:

Diagnostic tables
Tables that are like any other table in Derby. You can specify the diagnostic table name anywhere a normal table name is allowed.

Diagnostic table functions
Functions that are like any other function in Derby. Diagnostic table functions can accept zero or more arguments, depending on the table function that you use. You must use the SQL-defined table function syntax to access these functions.

The following table shows the types and names of the diagnostic table expressions in Derby.

Table 11. System diagnostic table expressions provided by Derby

Diagnostic Table Expression	Type of Expression
SYSCS_DIAG.CONTAINED_ROLES	Table function
SYSCS_DIAG.ERROR_LOG_READER	Table function
SYSCS_DIAG.ERROR_MESSAGES	Table
SYSCS_DIAG.LOCK_TABLE	Table

Diagnostic Table Expression	Type of Expression
SYSCS_DIAG.SPACE_TABLE	Table function
SYSCS_DIAG.STATEMENT_CACHE	Table
SYSCS_DIAG.STATEMENT_DURATION	Table function
SYSCS_DIAG.TRANSACTION_TABLE	Table

Restriction: If you reference a diagnostic table in a DDL statement or a compression procedure, Derby returns an exception.

SYSCS_DIAG.CONTAINED_ROLES diagnostic table function

The SYSCS_DIAG.CONTAINED_ROLES diagnostic table function returns all the roles contained within the specified role.

The argument that is passed to this table function should be the name of the role, specified as a string in quotes, or the special keyword CURRENT_ROLE, which indicates the current role in effect. For a definition of role containment, see "Syntax for roles" in GRANT statement.

For example:

```
SELECT * FROM TABLE (SYSCS_DIAG.CONTAINED_ROLES('READER')) AS T1
SELECT * FROM TABLE (SYSCS_DIAG.CONTAINED_ROLES(CURRENT_ROLE)) AS T2
```

All users can access this diagnostic table function, whether or not the database has authentication and SQL authorization enabled. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

The returned table has the column shown in the following table.

Table 12. Column returned by the SYSCS_DIAG.CONTAINED_ROLES table function

Column Name	Type	Length	Nullable	Contents
ROLEID	VARCHAR	128	false	The identifier of the role.

SYSCS_DIAG.ERROR_LOG_READER diagnostic table function

The SYSCS_DIAG.ERROR_LOG_READER diagnostic table function contains all the useful SQL statements that are in the `derby.log` file or a log file that you specify.

One use of this diagnostic table function is to determine the active transactions and the SQL statements in those transactions at a given point in time. For example, if a deadlock or lock timeout occurred, you can find the timestamp (timestampConstant) in the error log.

For a database for which authentication and SQL authorization are both enabled, only the database owner can access this diagnostic table function. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

To access the SYSCS_DIAG.ERROR_LOG_READER diagnostic table function, you must use the SQL table function syntax.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.ERROR_LOG_READER())
    AS T1
```

where T1 is a user-specified table name that is any valid identifier.

You can specify a log file name as an optional argument to the SYSCS_DIAG.ERROR_LOG_READER diagnostic table function. When you specify a log file name, the file name must be an expression whose data type maps to a Java string.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.ERROR_LOG_READER('myderbyerrors.log'))
    AS T1
```

Tip: By default, Derby log files contain only boot, shutdown, and error messages. See the `derby.stream.error.logSeverityLevel` property and the `derby.language.logStatementText` property for instructions on how to print more information to Derby log files. You can then query that information by using the SYSCS_DIAG.ERROR_LOG_READER diagnostic table function.

The returned table has the columns shown in the following table.

Table 13. Columns returned by the SYSCS_DIAG.ERROR_LOG_READER table function

Column Name	Type	Length	Nullable	Contents
TS	VARCHAR	26	false	The timestamp of the statement.
THREADID	VARCHAR	40	false	The thread name.
XID	VARCHAR	15	false	The transaction ID.
LCCID	VARCHAR	15	false	The connection ID.
DATABASE	VARCHAR	128	false	The database name.
DRDAID	VARCHAR	50	true	The DRDA ID for network server session.
LOGTEXT	LONG VARCHAR	32,700	false	The text of the statement or commit or rollback.

SYSCS_DIAG.ERROR_MESSAGES diagnostic table

The SYSCS_DIAG.ERROR_MESSAGES diagnostic table shows all of the *SQLState*s, locale-sensitive error messages, and exception severities for a Derby database.

You can reference the SYSCS_DIAG.ERROR_MESSAGES diagnostic table directly in a statement. For example:

```
SELECT * FROM SYSCS_DIAG.ERROR_MESSAGES
```

All users can access this diagnostic table, whether or not the database has authentication and SQL authorization enabled. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

The table has the columns shown in the following table.

Table 14. Columns in the SYSCS_DIAG.ERROR_MESSAGES table

Column Name	Type	Length	Nullable	Contents
SQL_STATE	VARCHAR	5	true	The *SQLState* of the *SQLException* (that is, the value returned by *SQLException.g*
MESSAGE	VARCHAR	32672	true	The error message (that is, the value returned by *SQLException.getMessage()*
SEVERITY	INTEGER	10	true	The Derby code for the severity (that is, the value returned by *SQLException.getError*

SYSCS_DIAG.LOCK_TABLE diagnostic table

The SYSCS_DIAG.LOCK_TABLE diagnostic table shows all of the locks that are currently held in the Derby database.

You can reference the SYSCS_DIAG.LOCK_TABLE diagnostic table directly in a statement. For example:

```
SELECT * FROM SYSCS_DIAG.LOCK_TABLE
```

All users can access this diagnostic table, whether or not the database has authentication and SQL authorization enabled. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

When the SYSCS_DIAG.LOCK_TABLE diagnostic table is referenced in a statement, a snapshot of the lock table is taken. A snapshot is used so that referencing the diagnostic table does not alter the normal timing and flow of the application. It is possible that some locks will be in a transition state when the snapshot is taken.

The table has the columns shown in the following table.

Table 15. Columns in the SYSCS_DIAG.LOCK_TABLE table

Column Name	Type	Length	Nullable	Contents
XID	VARCHAR	15	false	The transaction ID, which can be joined with the XID of the transaction table. See SYSCS_DIAG.TRA diagnostic table.
TYPE	VARCHAR	5	true	The type of lock, which can be either 'ROW', 'TABLE', or 'LATCH'.
MODE	VARCHAR	4	false	The mode of the lock. For a lock of type 'TABLE', the valid values are: *'S'* for shared lock *'U'* for update lock *'X'* for exclusive lock *'IS'* for intent shared lock

Column Name	Type	Length	Nullable	Contents
				'IX' for intent exclusive lock For a lock of type 'ROW', the valid values are: *'S'* for shared lock *'U'* for update lock *'X'* for exclusive lock For a lock of type 'LATCH', the only valid value is: *'X'* for exclusive lock
TABLENAME	VARCHAR	128	false	The name of the base table that the lock is for.
LOCKNAME	VARCHAR	20	false	The name of the lock.
STATE	VARCHAR	5	true	The state of the lock, which is either 'GRANT' or 'WAIT'.
TABLETYPE	VARCHAR	9	false	The type of the table. Valid values are: *'T'* for user table *'S'* for system table
LOCKCOUNT	VARCHAR	5	false	The internal lock count.
INDEXNAME	VARCHAR	128	true	Value is normally null. If it is non-null, a lock is held on the index.

SYSCS_DIAG.SPACE_TABLE diagnostic table function

The SYSCS_DIAG.SPACE_TABLE diagnostic table function shows the space usage of a particular table and its indexes.

You can use this diagnostic table function to determine if space might be saved by compressing the table and indexes.

All users can access this diagnostic table function, whether or not the database has authentication and SQL authorization enabled. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

To access the SYSCS_DIAG.SPACE_TABLE diagnostic table function, you must use the SQL table function syntax. You can invoke the table function in the following ways:

- If invoked with no arguments, the table function retrieves space information for all tables and indexes in the database.
- If invoked with one argument, a *tableName*, the table function retrieves information for the specified table in the current schema.
- If invoked with two arguments, a *schemaName* followed by a *tableName*, the table function retrieves information for the specified schema and table.

The returned table has the columns shown in the following table.

Table 16. Columns returned by the SYSCS_DIAG.SPACE_TABLE table function

Column Name	Type	Length	Nullable	Contents
CONGLOMERATENAME	VARCHAR	128	true	The name of the conglomerate, which is either the table name or the index name. (Unlike the SYSCONGLOMERATES column of the same name, table ID's do not appear here).
ISINDEX	SMALLINT	5	false	Is not zero if the conglomerate is an index, 0 otherwise.
NUMALLOCATEDPAGES	BIGINT	20	false	The number of pages actively linked into the table. The total number of pages in the file is the sum of NUMALLOCATEDPAGES + NUMFREEPAGES.
NUMFREEPAGES	BIGINT	20	false	The number of free pages that belong to the table. When a new page is to be linked into the table the system will move a page from the NUMFREEPAGES list to the NUMALLOCATEDPAGES list. The total number of pages in the file is the sum of NUMALLOCATEDPAGES + NUMFREEPAGES.
NUMUNFILLEDPAGES	BIGINT	20	false	The number of unfilled pages that belong to the table. Unfilled pages are allocated pages that are not completely full. Note that the number of unfilled pages is an estimate and is not exact. Running the same query twice can give different results on this column.
PAGESIZE	INTEGER	10	false	The size of the page in bytes for that conglomerate.
ESTIMSPACESAVING	BIGINT	20	false	The estimated space which could possibly be saved by compressing

Column Name	Type	Length	Nullable	Contents
				the conglomerate, in bytes.
TABLEID	CHAR	36	false	The id of the table to which the conglomerate belongs.

For example, use the following query to return the space usage for all of the user tables and indexes in the database:

```
SELECT sysschemas.schemaname, T2.*
    FROM
        SYS.SYSTABLES systabs, SYS.SYSSCHEMAS sysschemas,
        TABLE (SYSCS_DIAG.SPACE_TABLE()) AS T2
    WHERE systabs.tabletype = 'T'
    AND sysschemas.schemaid = systabs.schemaid
    AND systabs.tableid = T2.tableid;
```

where T2 is a user-specified table name that is any valid identifier.

Both the *schemaName* and the *tableName* arguments must be expressions whose data types map to Java strings. If the *schemaName* and the *tableName* are non-delimited identifiers, you must specify the names in uppercase.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.SPACE_TABLE('MYSCHEMA', 'MYTABLE'))
    AS T2
```

SYSCS_DIAG.STATEMENT_CACHE diagnostic table

The SYSCS_DIAG.STATEMENT_CACHE diagnostic table shows the contents of the SQL statement cache.

You can reference the SYSCS_DIAG.STATEMENT_CACHE diagnostic table directly in a statement. For example:

```
SELECT * FROM SYSCS_DIAG.STATEMENT_CACHE
```

For a database for which authentication and SQL authorization are both enabled, only the database owner can access this diagnostic table. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

The table has the columns shown in the following table.

Table 17. Columns in the SYSCS_DIAG.STATEMENT_CACHE table

Column Name	Type	Length	Nullable	Contents
ID	CHAR	36	false	The internal identifier of the compiled statement.
SCHEMANAME	VARCHAR	128	true	The schema the statement was compiled in.
SQL_TEXT	VARCHAR	32,672	false	The text of the statement.
UNICODE	BOOLEAN	1	false	Always *true*

Column Name	Type	Length	Nullable	Contents
VALID	BOOLEAN	1	false	*true* (the statement is currently valid) *false* (the statement is not currently valid)
COMPILED_AT	TIMESTAMP	29	true	The time the statement was compiled. Requires statistics timing to be enabled (see SYSCS_UTIL.SYSCS_SET system procedure).

SYSCS_DIAG.STATEMENT_DURATION diagnostic table function

You can use the SYSCS_DIAG.STATEMENT_DURATION diagnostic table function to analyze the execution duration of the useful SQL statements in the `derby.log` file or a log file that you specify.

You can also use this diagnostic table function to get an indication of where the bottlenecks are in the JDBC code for an application.

For a database for which authentication and SQL authorization are both enabled, only the database owner can access this diagnostic table function. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

To access the SYSCS_DIAG.STATEMENT_DURATION diagnostic table function, you must use the SQL table function syntax.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.STATEMENT_DURATION())
    AS T1
```

where T1 is a user-specified table name that is any valid identifier.

Restriction: For each transaction ID, a row is not returned for the last statement with that transaction ID. Transaction IDs change within a connection after a commit or rollback, if the transaction that just ended modified data.

You can specify a log file name as an optional argument to the SYSCS_DIAG.STATEMENT_DURATION diagnostic table function. When you specify a log file name, the file name must be an expression whose data type maps to a Java string.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.STATEMENT_DURATION('somederby.log'))
    AS T1
```

Tip: By default Derby log files contain only boot, shutdown, and error messages. See the `derby.stream.error.logSeverityLevel` property and the `derby.language.logStatementText` property for instructions on how to print more information to Derby log files. You can then query that information by using the SYSCS_DIAG.STATEMENT_DURATION diagnostic table function.

The returned table has the columns shown in the following table.

Table 18. Columns returned by the SYSCS_DIAG.STATEMENT_DURATION table function

Column Name	Type	Length	Nullable	Contents
TS	VARCHAR	26	false	The timestamp of the statement.
THREADID	VARCHAR	80	false	The thread name.
XID	VARCHAR	15	false	The transaction ID.
LOGTEXT	LONG VARCHAR	32,700	true	The text of the statement or commit or rollback.
DURATION	VARCHAR	10	false	The duration, in milliseconds, of the statement.

SYSCS_DIAG.TRANSACTION_TABLE diagnostic table

The SYSCS_DIAG.TRANSACTION_TABLE diagnostic table shows all of the transactions that are currently in the database.

You can reference the SYSCS_DIAG.TRANSACTION_TABLE diagnostic table directly in a statement. For example:

```
SELECT * FROM SYSCS_DIAG.TRANSACTION_TABLE
```

When the SYSCS_DIAG.TRANSACTION_TABLE diagnostic table is referenced in a statement, a snapshot of the transaction table is taken. A snapshot is used so that referencing the diagnostic table does not alter the normal timing and flow of the application. It is possible that some transactions will be in a transition state when the snapshot is taken.

For a database for which authentication and SQL authorization are both enabled, only the database owner can access this diagnostic table. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

The table has the columns shown in the following table.

Table 19. Columns in the SYSCS_DIAG.TRANSACTION_TABLE table

Column Name	Type	Length	Nullable	Contents
XID	VARCHAR	15	false	The transaction id, which can be joined with the lock table virtual table's XID. See SYSCS_DIAG.LC diagnostic table.
GLOBAL_XID	VARCHAR	140	true	The global transaction ID, set only if this transaction is a participant in a distributed transaction.
USERNAME	VARCHAR	128	true	The user name (the default value is APP).

Column Name	Type	Length	Nullable	Contents
				May appear null if the transaction is started by Derby.
TYPE	VARCHAR	30	false	Either 'UserTransaction' or an internal transaction spawned by Derby.
STATUS	VARCHAR	8	false	Either 'IDLE' or 'ACTIVE'. A transaction is IDLE only when it is first created or right after it commits. Any transaction that holds or has held any resource in the database is ACTIVE. Accessing the TransactionTable virtual table without using the class alias will not activate the transaction.
FIRST_INSTANT	VARCHAR	20	true	If null, this is a read-only transaction. If not null, this is the first log record instant written by the transaction.
SQL_TEXT	VARCHAR	32,672	true	If null, this transaction is currently not being executed in the database. If not null, this is the SQL statement currently being executed in the database.

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 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.

The following table shows the promotion of data types in expressions.

Table 20. 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, i INTEGER, de DECIMAL);
0 rows inserted/updated/deleted
INSERT INTO mytable (r, d) values (3.4028236E38, 3.4028235E38);
ERROR 22003: The resulting value 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 (i) VALUES (1.09e0);
1 row inserted/updated/deleted
SELECT i 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:

```
INSERT INTO mytable (de) VALUES (55555555556666666666);
ERROR 22003: The resulting value is outside the range for the
data type DECIMAL/NUMERIC(5,2).
```

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 (i) 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

The tables in this section show valid assignments and comparisons between Derby data types.

Sorting and ordering of character data is controlled by the collation specified for a database when it is created, as well as the locale of the database. For details, see *collation=collation* attribute and *territory=ll_CC* attribute, as well as the sections "Creating a database with locale-based collation", "Creating a case-insensitive database", and "Character-based collation in Derby" in the *Derby Developer's Guide*.

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

Table 21. Assignments allowed by Derby

Types	BOOLEAN	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	XML	User-defined type
BOOLEAN	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SMALLINT	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-
INTEGER	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-

Types	BOOLEAN	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	XML	User-defined type
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 FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	-	-	-	-
CLOB	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-	-
DATE	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-	-	-
TIME	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-	-	-

Types	BOOLEAN	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	XML	User-defined type
TIMESTAMP	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	Y	-	-
XML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-
User-defined type	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y

A value of a user-defined type can be assigned to a value of any supertype of that user-defined type. However, no explicit casts of user-defined types are allowed.

The following table displays valid comparisons between data types in Derby. A "Y" indicates that the comparison is allowed.

Table 22. Comparisons allowed by Derby

Types	BOOLEAN	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	XML	User-defined type
BOOLEAN	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SMALLINT	-	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	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Types	BOOLEAN	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	XML	User-defined type
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DATE	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-	-	-
TIME	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-	-	-
TIMESTAMP	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	Y	-	-
XML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
User-defined type	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

BIGINT data type

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.

Example

```
9223372036854775807
```

BLOB data type

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, and 1024*1024*1024 respectively.

Note: Length is specified in bytes for BLOB.

Syntax

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

Default

A BLOB without a specified length is defaulted to two gigabytes (2,147,483,647).

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 Mapping of java.sql.Blob and java.sql.Clob interfaces.

Examples

```
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 comparisons)
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;
```

Using an INSERT statement to put BLOB data into a table has some limitations if you need to cast a long string constant to a BLOB. (See String limitations.) You may be better off using a binary stream, as in the following code fragment.

```
        String url = "jdbc:derby:blobby;create=true";
        Connection conn = DriverManager.getConnection(url);

        Statement s = conn.createStatement();
        s.executeUpdate(
            "CREATE TABLE images (id INT, img BLOB)");

        // - first, create an input stream
        InputStream fin = new FileInputStream("image.jpg");

        PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO images VALUES (?, ?)");
        ps.setInt(1, 1477);

        // - set the value of the input parameter to the input stream
        ps.setBinaryStream(2, fin);
        ps.execute();

        // --- reading the columns
        ResultSet rs = s.executeQuery(
            "SELECT img FROM images WHERE id = 1477");
        byte buff[] = new byte[1024];

        while (rs.next()) {
            Blob ablob = rs.getBlob(1);
            File newfile = new File("newimage.jpg");

            InputStream is = ablob.getBinaryStream();

            FileOutputStream fos =
                new FileOutputStream(newfile);

            for (int b = is.read(buff); b != -1; b = is.read(buff)) {
                fos.write(buff, 0, b);
            }

            is.close();
            fos.close();
        }
        s.close();
        ps.close();
        rs.close();
        conn.close();
```

BOOLEAN data type

BOOLEAN provides 1 byte of storage for logical values.

Syntax

```
BOOLEAN
```

Corresponding compile-time Java type

java.lang.Boolean

JDBC metadata type (java.sql.Types)

BOOLEAN

Legal values

The legal values are *true*, *false*, and *null*. BOOLEAN values can be cast to and from character typed values. For comparisons and ordering operations, *true* sorts higher than *false*.

Examples

```
values true
```

```
values false
values cast (null as boolean)
```

CHAR data type

CHAR provides for fixed-length storage of strings.

Syntax

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

length is an unsigned integer literal designating the length in bytes. The default *length* for a CHAR is 1, and the maximum size of *length* is 254.

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 comparison boolean 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.

Examples

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

```
-- create a table with a CHAR field
CREATE TABLE STATUS (
    STATUSCODE CHAR(2) NOT NULL
        CONSTRAINT PK_STATUS PRIMARY KEY,
    STATUSDESC VARCHAR(40) NOT NULL
);
```

CHAR FOR BIT DATA data type

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.)

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.

Example

```
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 data type

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}] ) ]
```

Default

A CLOB without a specified length is defaulted to two giga characters (2,147,483,647).

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 Mapping of java.sql.Blob and java.sql.Clob interfaces.

Example

```
        String url = "jdbc:derby:clobberyclob;create=true";
        Connection conn = DriverManager.getConnection(url);

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

        // - first, create an input stream
        InputStream fis = new FileInputStream("asciifile.txt");

        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, fis);
        ps.execute();

        // --- reading the columns back
        ResultSet rs = s.executeQuery(
            "SELECT text FROM documents WHERE id = 1477");

        while (rs.next()) {
            Clob aclob = rs.getClob(1);
            InputStream ip = aclob.getAsciiStream();

            for (int c = ip.read(); c != -1; c = ip.read()) {
                System.out.print((char)c);
            }

            ip.close();
        }
        s.close();
        ps.close();
        rs.close();
        conn.close();
```

DATE data type

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 data type

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.

Integer constants too big for BIGINT are made DECIMAL constants.

Corresponding compile-time Java type

java.math.BigDecimal

JDBC metadata type (java.sql.Types)

DECIMAL

Examples

```
VALUES 123.456

VALUES 0.001
```

DOUBLE data type

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

See DOUBLE PRECISION data type.

Syntax

```
DOUBLE
```

DOUBLE PRECISION data type

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:

- Largest negative DOUBLE value: -1.7976931348623157E+308
- Largest positive DOUBLE value: 1.7976931348623157E+308
- Smallest negative normalized DOUBLE value: -2.2250738585072014E-308
- Smallest positive normalized DOUBLE value: 2.2250738585072014E-308
- Smallest negative denormalized DOUBLE value: -4.9E-324
- Smallest positive denormalized DOUBLE value: 4.9E-324

These limits are the same as 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.

Derby normalizes -0.0 to positive 0.0.

Derby throws an exception if an operation calculates or tries to store a value of NaN, positive infinity, or negative infinity, as defined by the *IEEE 754 Standard for Binary Floating-Point Arithmetic* and as represented with named constants in the Java programming language (for example, `Double.NaN`).

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 data type

The FLOAT data type is an alias for a REAL or DOUBLE PRECISION data type, depending on the precision you specify.

See REAL data type and DOUBLE PRECISION data type.

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 data type

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.

Examples

```
3453
425
```

LONG VARCHAR data type

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 cannot 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 data type

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 cannot specify a maximum length when creating columns of this type.

Syntax

```
LONG VARCHAR FOR BIT DATA
```

JDBC metadata type (java.sql.Types)

LONGVARBINARY

NUMERIC data type

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

See DECIMAL data type.

Syntax

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

Corresponding compile-time Java type

java.math.BigDecimal

JDBC metadata type (java.sql.Types)

NUMERIC

Examples

```
123.456
.001
```

REAL data type

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:

- Largest negative REAL value: -3.4028235E+38
- Largest positive REAL value: 3.4028235E+38
- Smallest negative normalized REAL value: -1.17549435E-38
- Smallest positive normalized REAL value: 1.17549435E-38
- Smallest negative denormalized REAL value: -1.4E-45
- Smallest positive denormalized REAL value: 1.4E-45

These limits are the same as 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 underflows.

Derby normalizes -0.0 to positive 0.0.

Derby throws an exception if an operation calculates or tries to store a value of NaN, positive infinity, or negative infinity, as defined by the *IEEE 754 Standard for Binary Floating-Point Arithmetic* and as represented with named constants in the Java programming language (for example, `Double.NaN`).

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 data type

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 data type

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 data type

TIMESTAMP provides for storage of a combined DATE and TIME value. It permits a fractional-seconds value of up to nine digits.

See DATE data type and TIME data type.

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[.nnnnnnnnn]
yyyy-mm-dd-hh.mm.ss[.nnnnnnnnn]
```

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 must have two digits. Nanoseconds, if present, may have between one and nine digits. The year, month, and day components must be positive integers.

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')
VALUES TIMESTAMP('1962-09-23-03:23:34.987654321')
```

User-defined types

Derby allows you to create user-defined types. A user-defined type is a serializable Java class whose instances are stored in columns. The class must implement the *java.io.Serializable* interface.

For information on creating and removing types, see CREATE TYPE statement and DROP TYPE statement. See GRANT statement and REVOKE statement for information on usage privileges for types.

For information on writing the Java classes that implement user-defined types, see "Programming user-defined types" in the *Derby Developer's Guide*.

VARCHAR data type

VARCHAR provides for variable-length storage of strings.

Syntax

```
{ VARCHAR | CHAR VARYING | CHARACTER VARYING }(length)
```

length is an unsigned integer constant. 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 comparison boolean 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.

If you use a VARCHAR as a key column of an index, limit the maximum size of the VARCHAR to no more than half the page size to prevent inserts from failing. See "Page size and key size" in CREATE INDEX statement for details.

VARCHAR FOR BIT DATA data type

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 (for example, 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.

XML data type

The XML data type is used for Extensible Markup Language (XML) documents.

The XML data type is used:

- To store XML documents that conform to the SQL/XML definition of a well-formed XML(DOCUMENT(ANY)) value.
- Transiently, for XML(SEQUENCE) values that might not be well-formed XML(DOCUMENT(ANY)) values.

Because none of the JDBC-side support for SQL/XML is implemented in Derby, it is not possible to bind directly into an XML value or to retrieve an XML value directly from a result set using JDBC. Instead, you must bind and retrieve the XML data as Java strings or character streams by explicitly specifying the appropriate XML operators, XMLPARSE and XMLSERIALIZE, as part of your SQL queries.

Syntax

```
XML
```

Corresponding compile-time Java type

None.

The Java type for XML values is *java.sql.SQLXML*. However, the *java.sql.SQLXML* type is not supported by Derby.

JDBC metadata type (java.sql.Types)

None.

The metadata type for XML values is SQLXML. However, the SQLXML type is not supported by Derby.

To retrieve XML values from a Derby database using JDBC, use the XMLSERIALIZE operator in the SQL query. For example:

```
SELECT XMLSERIALIZE (xcol as CLOB) FROM myXmlTable
```

Then retrieve the XML value by using the *getXXX* method that corresponds to the target serialization type, in this example CLOB data types.

To store an XML value into a Derby database using JDBC, use the XMLPARSE operator in the SQL statement. For example:

```
INSERT INTO myXmlTable(xcol) VALUES XMLPARSE(
    DOCUMENT CAST (? AS CLOB) PRESERVE WHITESPACE)
```

Then use any of the *setXXX* methods that are compatible with *String* types. In this example, use the *PreparedStatement.setString* or *PreparedStatement.setCharacterStream* method calls to bind the operator.

See "XML data types and operators" in the *Derby Developer's Guide* for more information.

Argument matching

When you declare a function or procedure using CREATE FUNCTION/PROCEDURE, Derby does not verify whether a matching Java method exists. Instead, Derby looks for a matching method only when you invoke the function or procedure in a later SQL statement.

At that time, Derby searches for a public, static method having the class and method name declared in the EXTERNAL NAME clause of the earlier CREATE FUNCTION/PROCEDURE statement. Furthermore, the Java types of the method's arguments and return value must match the SQL types declared in the CREATE FUNCTION/PROCEDURE statement. The following may happen:

- **Success** - If exactly one Java method matches, then Derby invokes it.
- **Ambiguity** - Derby raises an error if more than one method matches.
- **Failure** - Derby also raises an error if no method matches.

A procedure or function that takes varargs must resolve to a varargs Java method.

In mapping SQL data types to Java data types, Derby considers the following kinds of matches:

- **Primitive match** - Derby looks for a primitive Java type corresponding to the SQL type. For instance, SQL INTEGER matches Java *int*.
- **Wrapper match** - Derby looks for a wrapper class in the *java.lang* or *java.sql* packages corresponding to the SQL type. For instance, SQL INTEGER matches *java.lang.Integer*. For a user-defined type (UDT), Derby looks for the UDT's external name class.
- **Array match** - For OUT and INOUT procedure arguments, Derby looks for an array of the corresponding primitive or wrapper type. For instance, an OUT procedure argument of type SQL INTEGER matches *int[]* and *Integer[]*.
- **ResultSet match** - If a procedure is declared to return *n* RESULT SETS, Derby looks for a method whose last *n* arguments are of type *java.sql.ResultSet[]*.

Derby resolves function and procedure invocations as follows:

- **Function** - Derby looks for a method whose argument and return types are *primitive match*es or *wrapper match*es for the function's SQL arguments and return value.
- **Procedure** - Derby looks for a method which returns void and whose argument types match as follows:
  - *IN* - Method arguments are *primitive match*es or *wrapper match*es for the procedure's IN arguments.
  - *OUT and INOUT* - Method arguments are *array match*es for the procedure's OUT and INOUT arguments.

  In addition, if the procedure returns *n* RESULT SETS, then the last *n* arguments of the Java method must be of type *java.sql.ResultSet[]*.

Derby provides a tool, `SignatureChecker`, which can identify any SQL functions or procedures in a database that do not follow these argument matching rules. See the *Derby Tools and Utilities Guide* for details.

Example of argument matching

The following function...

```
CREATE FUNCTION TO_DEGREES
( RADIANS DOUBLE )
RETURNS DOUBLE
```

```
PARAMETER STYLE JAVA
NO SQL LANGUAGE JAVA
EXTERNAL NAME 'example.MathUtils.toDegrees'
```

...would match all of the following methods:

```
public static double toDegrees( double arg ) {...}
public static Double toDegrees( double arg ) {...}
public static double toDegrees( Double arg ) {...}
public static Double toDegrees( Double arg ) {...}
```

Note that Derby would raise an exception if it found more than one matching method.

Mapping SQL data types to Java data types

The following table shows how Derby maps specific SQL data types to Java data types.

Table 23. SQL and Java type correspondence

SQL Type	Primitive Match	Wrapper Match
BOOLEAN	*boolean*	*java.lang.Boolean*
SMALLINT	*short*	*java.lang.Integer*
INTEGER	*int*	*java.lang.Integer*
BIGINT	*long*	*java.lang.Long*
DECIMAL	None	*java.math.BigDecimal*
NUMERIC	None	*java.math.BigDecimal*
REAL	*float*	*java.lang.Float*
DOUBLE	*double*	*java.lang.Double*
FLOAT	*double*	*java.lang.Double*
CHAR	None	*java.lang.String*
VARCHAR	None	*java.lang.String*
LONG VARCHAR	None	*java.lang.String*
CHAR FOR BIT DATA	*byte[]*	None
VARCHAR FOR BIT DATA	*byte[]*	None
LONG VARCHAR FOR BIT DATA	*byte[]*	None
CLOB	None	*java.sql.Clob*
BLOB	None	*java.sql.Blob*
DATE	None	*java.sql.Date*
TIME	None	*java.sql.Time*
TIMESTAMP	None	*java.sql.Timestamp*
XML	None	None
User-defined type	None	Underlying Java class

SQL reserved words

This section lists all the Derby reserved words, including those in the SQL 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 SQL identifiers.

ADD
ALL
ALLOCATE
ALTER
AND
ANY
ARE
AS
ASC
ASSERTION
AT
AUTHORIZATION
AVG
BEGIN
BETWEEN
BIGINT
BIT
BOOLEAN
BOTH
BY
CALL
CASCADE
CASCADED
CASE
CAST
CHAR
CHARACTER
CHARACTER_LENGTH
CHECK
CLOSE
COALESCE
COLLATE
COLLATION
COLUMN
COMMIT
CONNECT
CONNECTION
CONSTRAINT
CONSTRAINTS
CONTINUE
CONVERT
CORRESPONDING
CREATE
CROSS
CURRENT
CURRENT_DATE
CURRENT_ROLE
CURRENT_TIME

CURRENT_TIMESTAMP
CURRENT_USER
CURSOR
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DEFERRABLE
DEFERRED
DELETE
DESC
DESCRIBE
DIAGNOSTICS
DISCONNECT
DISTINCT
DOUBLE
DROP
ELSE
END
END-EXEC
ESCAPE
EXCEPT
EXCEPTION
EXEC
EXECUTE
EXISTS
EXPLAIN
EXTERNAL
FALSE
FETCH
FIRST
FLOAT
FOR
FOREIGN
FOUND
FROM
FULL
FUNCTION
GET
GETCURRENTCONNECTION
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
LEADING
LEFT
LIKE
LOWER
LTRIM
MATCH
MAX
MIN
MINUTE
NATIONAL
NATURAL
NCHAR
NVARCHAR
NEXT
NO
NONE
NOT
NULL
NULLIF
NUMERIC
OF
ON
ONLY
OPEN
OPTION
OR
ORDER
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
TRANSACTION
TRANSLATE
TRANSLATION
TRIM
TRUE
UNION
UNIQUE
UNKNOWN
UPDATE
UPPER
USER
USING
VALUES
VARCHAR
VARYING
VIEW
WHENEVER
WHERE
WINDOW
WITH
WORK
WRITE
XML
XMLEXISTS
XMLPARSE
XMLQUERY
XMLSERIALIZE
YEAR

Derby support for SQL:2011 features

The SQL:2011 standard puts features into two categories, mandatory and optional.

In the tables that follow, the support status of each feature is indicated as follows:

- **Yes**: The feature is supported.
- **Yes***: The feature is supported (for example, through JDBC) but not according to the SQL standard. See Note for details.
- **Partial**: The feature is partially supported.
- **No**: The feature is unsupported.

Derby supports many features of the SQL:2011 standard. Most are in Part 2 (Foundation), but some are in other parts. The supported optional features with the prefix J are in Part 13 (JRT). The supported optional features with the prefix X are in Part 14 (SQL/XML).

SQL:2011 features not supported by Derby lists the features in Part 2 that Derby does not support.

Support for mandatory features

The following tables describe support for SQL:2011 mandatory features.

Table 24. E011: Numeric data types

Feature ID	Feature Name	SQL:2011 Mandatory
E011-01	INTEGER and SMALLINT data types (including all spellings)	Yes
E011-02	REAL, DOUBLE PRECISION, and FLOAT data types	Yes
E011-03	DECIMAL and NUMERIC data types	Yes
E011-04	Arithmetic operators	Yes
E011-05	Numeric comparison	Yes
E011-06	Implicit casting among the numeric data types	Yes

Table 25. E021: Character data types

Feature ID	Feature Name	SQL:2011 Mandatory	Note
E021-01	CHARACTER data type (including all its spellings)	Yes	None
E021-02	CHARACTER VARYING data type (including all its spellings)	Yes	None
E021-03	Character literals	Yes	None
E021-04	CHARACTER_LENGTH function	Yes*	Called LENGTH. `{fn LENGTH(...)}` is according to JDBC specification.

Feature ID	Feature Name	SQL:2011 Mandatory	Note
E021-05	OCTET_LENGTH function	No	None
E021-06	SUBSTRING function	Yes*	Called SUBSTR. `{fn SUBSTRING(....)}` is according to JDBC specification.
E021-07	Character concatenation	Yes	None
E021-08	UPPER and LOWER functions	Yes	None
E021-09	TRIM function	Yes	None
E021-10	Implicit casting among the character data types	Yes	None
E021-11	POSITION function	Yes*	Called LOCATE. `{fn LOCATE(...)}` is according to JDBC specification.
E021-12	Character comparison	Yes	None

Table 26. E031: Identifiers

Feature ID	Feature Name	SQL:2011 Mandatory
E031-01	Delimited identifiers	Yes
E031-02	Lower case identifiers	Yes
E031-03	Trailing underscore	Yes

Table 27. E051: Basic query specification

Feature ID	Feature Name	SQL:2011 Mandatory
E051-01	SELECT DISTINCT	Yes
E051-02	GROUP BY clause	Yes
E051-04	GROUP BY can contain columns not in select-list	Yes
E051-05	Select list items can be renamed	Yes
E051-06	HAVING clause	Yes
E051-07	Qualified * in select list	Yes
E051-08	Correlation names in the FROM clause	Yes
E051-09	Rename columns in the FROM clause	Yes

Table 28. E061: Basic predicates and search conditions

Feature ID	Feature Name	SQL:2011 Mandatory
E061-01	Comparison predicate	Yes
E061-02	BETWEEN predicate	Yes
E061-03	IN predicate with list of values	Yes
E061-04	LIKE predicate	Yes
E061-05	LIKE predicate: ESCAPE clause	Yes
E061-06	NULL predicate	Yes
E061-07	Quantified comparison predicate	Yes
E061-08	EXISTS predicate	Yes
E061-09	Subqueries in comparison predicate	Yes
E061-11	Subqueries in IN predicate	Yes
E061-12	Subqueries in quantified comparison predicate	Yes
E061-13	Correlated subqueries	Yes
E061-14	Search condition	Yes

Table 29. E071: Basic query expressions

Feature ID	Feature Name	SQL:2011 Mandatory
E071-01	UNION DISTINCT table operator	Yes
E071-02	UNION ALL table operator	Yes
E071-03	EXCEPT DISTINCT table operator	Yes
E071-05	Columns combined via table operators need not have exactly the same data type	Yes
E071-06	Table operators in subqueries	Yes

Table 30. E081: Basic privileges

Feature ID	Feature Name	SQL:2011 Mandatory
E081-01	SELECT privilege at the table level	Yes
E081-02	DELETE privilege	Yes
E081-03	INSERT privilege at the table level	Yes
E081-04	UPDATE privilege at the table level	Yes
E081-05	UPDATE privilege at the column level	Yes
E081-06	REFERENCES privilege at the table level	Yes
E081-07	REFERENCES privilege at the column level	Yes
E081-08	WITH GRANT OPTION	No

Feature ID	Feature Name	SQL:2011 Mandatory
E081-09	USAGE privilege	No
E081-10	EXECUTE privilege	Yes

Table 31. E091: Set functions

Feature ID	Feature Name	SQL:2011 Mandatory
E091-01	AVG	Yes
E091-02	COUNT	Yes
E091-03	MAX	Yes
E091-04	MIN	Yes
E091-05	SUM	Yes
E091-06	ALL quantifier	Yes
E091-07	DISTINCT qualifier	Yes

Table 32. E101: Basic data manipulation

Feature ID	Feature Name	SQL:2011 Mandatory	Note
E101-01	INSERT statement	Yes	None
E101-03	Searched UPDATE statement	Partial	*correlation name* not supported
E101-04	Searched DELETE statement	Partial	*correlation name* not supported

Table 33. E121: Basic cursor support (through JDBC)

Feature ID	Feature Name	SQL:2011 Mandatory	Note
E121-01	Declare cursor	No	None
E121-02	ORDER BY columns need not be in select list	Yes	None
E121-03	Value expressions in ORDER BY clause	Yes	None
E121-06	Positioned UPDATE statement	Partial	*correlation name* not supported
E121-07	Positioned DELETE statement	Partial	*correlation name* not supported
E121-08	CLOSE statement	No	None
E121-10	FETCH statement	No	None
E121-17	WITH HOLD cursors	No	None

Table 34. E141: Basic integrity constraints

Feature ID	Feature Name	SQL:2011 Mandatory
E141-01	NOT NULL constraints	Yes
E141-02	UNIQUE constraints of NOT NULL columns	Yes
E141-03	PRIMARY KEY constraints	Yes
E141-04	Basic FOREIGN KEY constraint with the NO ACTION default	Yes
E141-06	CHECK constraints	Yes
E141-07	Column defaults	Yes
E141-08	NOT NULL inferred on PRIMARY KEY	Yes
E141-10	Names in a foreign key can be specified in any order	Yes

Table 35. E151: Transaction support

Feature ID	Feature Name	SQL:2011 Mandatory	Note
E151-01	COMMIT statement	Yes*	Through JDBC `Connection.commit`, `ij` supports COMMIT statement
E151-02	ROLLBACK statement	Yes*	Through JDBC `Connection.rollback`, `ij` supports ROLLBACK statement

Table 36. E152: Basic SET TRANSACTION statement

Feature ID	Feature Name	SQL:2011 Mandatory	Note
E152-01	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	Yes*	SET [CURRENT] ISOLATION SERIALIZABLE. `Connection.se` `Connection.TRANSACTION_SERIALIZABL` is according to JDBC specification.
E152-02	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	Yes*	No SQL syntax. `Connection.setReadWrit` is according to JDBC specification.

Table 37. F031: Basic schema manipulation

Feature ID	Feature Name	SQL:2011 Mandatory
F031-01	CREATE TABLE statement to create persistent base tables	Yes

Feature ID	Feature Name	SQL:2011 Mandatory
F031-02	CREATE VIEW statement	Yes
F031-03	GRANT statement	Yes
F031-04	ALTER TABLE statement: ADD COLUMN clause	Yes
F031-13	DROP TABLE statement: RESTRICT clause	Yes (implicit)
F031-16	DROP VIEW statement: RESTRICT clause	Yes (implicit)
F031-19	REVOKE statement: RESTRICT clause	Yes

Table 38. F041: Basic joined tables

Feature ID	Feature Name	SQL:2011 Mandatory
F041-01	Inner join (but not necessarily the INNER keyword)	Yes
F041-02	INNER keyword	Yes
F041-03	LEFT OUTER JOIN	Yes
F041-04	RIGHT OUTER JOIN	Yes
F041-05	Outer joins can be nested	Yes
F041-07	The inner table in a left or right outer join can also be used in an inner join	No
F041-08	All comparison operators are supported (rather than just =)	Yes

Table 39. F051: Basic date and time

Feature ID	Feature Name	SQL:2011 Mandatory	Note
F051-01	DATE data type (including DATE literal)	Yes*	DATE literal is implemented as built-in function. `{d 'yyyy-mm-ff'}` is according to JDBC specification.
F051-02	TIME data type (including TIME literal) with fractional seconds precision of 0	Yes*	TIME literal is implemented as built-in function. No precision in datatype. `{t' hh:mm:ss'}` is according to JDBC specification.
F051-03	TIMESTAMP data type (including TIMESTAMP literal) with fractional seconds precision of 0 and 6	Yes*	TIMESTAMP literal is implemented as built-in function. No precision in datatype. No timezone in datatype.

Feature ID	Feature Name	SQL:2011 Mandatory	Note
			`{ts 'yyyy-mm-dd hh:mm:ss.f...'}` is according to JDBC specification.
F051-04	Comparison predicate on DATE, TIME, and TIMESTAMP data types	Yes	None
F051-05	Explicit CAST between datetime types and character types	Yes	None
F051-06	CURRENT_DATE	Yes*	No time zone in datetime value expression
F051-07	LOCALTIME	Yes*	`{fn CURTIME()}` is according to JDBC specification
F051-08	LOCALTIMESTAMP	No	None

Table 40. F131: Grouped operations

Feature ID	Feature Name	SQL:2011 Mandatory
F131-01	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	Yes
F131-02	Multiple tables supported in queries with grouped views	Yes
F131-03	Set functions supported in queries with grouped views	Yes
F131-04	Subqueries with GROUP BY and HAVING clauses and grouped views	Yes
F131-05	Single row SELECT with GROUP BY and HAVING clauses and grouped views	Yes

Table 41. F261: CASE expression

Feature ID	Feature Name	SQL:2011 Mandatory
F261-01	Simple CASE	Yes
F261-02	Searched CASE	Yes
F261-03	NULLIF function	Yes
F261-04	COALESCE function	Yes

Table 42. F311: Schema definition statement

Feature ID	Feature Name	SQL:2011 Mandatory
F311-01	Create schema	Yes
F311-02	CREATE TABLE for persistent base tables	Yes
F311-03	CREATE VIEW	Yes
F311-04	CREATE VIEW: WITH CHECK OPTION	No
F311-05	GRANT statement	Yes

Table 43. T321: Basic SQL invoked routines

Feature ID	Feature Name	SQL:2011 Mandatory
T321-01	User-defined functions with no overloading	Yes
T321-02	User-defined stored procedures with no overloading	Yes
T321-03	Function invocation	Yes
T321-04	CALL statement	Yes
T321-05	RETURN statement	No

Table 44. Miscellaneous mandatory features

Feature ID	Feature Name	SQL:2011 Mandatory
E111	Single row select statement	Yes
E131	Null value support (nulls in lieu of values)	Yes
E161	SQL comments using leading double minus	Yes
E171	SQLSTATE support	Yes
F201	CAST function	Yes
F221	Explicit defaults	Yes
F471	Scalar subquery values	Yes
F481	Expanded NULL predicate	Yes
T631	IN predicate with one list element	Yes

Support for optional features

The following tables show Derby support for SQL:2011 optional features.

Table 45. F111: Isolation levels other than SERIALIZABLE

Feature ID	Feature Name	SQL:2011 Optional
F111-01	READ UNCOMMITTED isolation level	Yes
F111-02	READ COMMITTED isolation level	Yes

Feature ID	Feature Name	SQL:2011 Optional
F111-03	REPEATABLE READ isolation level	Yes

Table 46. F302: INTERSECT table operator

Feature ID	Feature Name	SQL:2011 Optional
F302-01	INTERSECT DISTINCT table operator	Yes
F302-02	INTERSECT ALL table operator	Yes

Table 47. F381: Extended schema manipulation

Feature ID	Feature Name	SQL:2011 Optional
F381-01	ALTER TABLE statement: ALTER COLUMN clause	Partial
F381-02	ALTER TABLE statement: ADD CONSTRAINT clause	Partial
F381-03	ALTER TABLE statement: DROP CONSTRAINT clause	Yes

Table 48. F401: Extended joined table

Feature ID	Feature Name	SQL:2011 Optional
F401-01	NATURAL JOIN	Yes
F401-02	FULL OUTER JOIN	No
F401-04	CROSS JOIN	Yes

Table 49. F831: Full cursor update

Feature ID	Feature Name	SQL:2011 Optional	Note
F831-01	Updatable scrollable cursors	Partial	Insensitive result set cursors
F831-02	Updatable ordered cursors	No	None

Table 50. T041: Basic LOB data type support

Feature ID	Feature Name	SQL:2011 Optional	Note
T041-01	BLOB data type	Yes	None
T041-02	CLOB data type	Yes	None
T041-03	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	Yes*	Not standard SQL syntax. See notes on features E021-04, E021-06, E021-09 and E021-11
T041-04	Concatenation of LOB data types	Yes	None

Feature ID	Feature Name	SQL:2011 Optional	Note
T041-05	LOB locator: non-holdable	No	None

Table 51. T211: Basic trigger capability

Feature ID	Feature Name	SQL:2011 Optional	Note
T211-01	Triggers activated on UPDATE, INSERT, or DELETE of one base table	Yes	None
T211-02	BEFORE triggers	Yes*	Need to specify non-standard ON CASCADE BEFORE. Before triggers cannot have INSERT, UPDATE or DELETE statements as their action
T211-03	AFTER triggers	Yes	None
T211-04	FOR EACH ROW triggers	Yes	None
T211-05	Ability to specify a search condition that shall be True before the trigger is invoked	Yes	None
T211-06	Support for run-time rules for the interaction of triggers and constraints	No	None
T211-07	TRIGGER privilege	Yes	None
T211-08	Multiple triggers for the same event are executed in the order in which they were created in the catalog	Yes	None

Table 52. Miscellaneous optional features

Feature ID	Feature Name	SQL:2011 Optional	Note
F033	ALTER TABLE statement: DROP COLUMN clause	Yes	None
F200	TRUNCATE TABLE statement	Yes	None
F262	Extended CASE expression	Yes	None
F263	Comma-separated predicates in simple CASE expression	Yes	None
F281	LIKE enhancements	Yes	None
F304	EXCEPT ALL table operator	Yes	None
F312	MERGE statement	Yes	None

Feature ID	Feature Name	SQL:2011 Optional	Note
F313	Enhanced MERGE statement	Yes	None
F314	MERGE statement with DELETE branch	Yes	None
F382	Alter column data type	Partial	You can alter only VARCHAR, VARCHAR FOR BIT DATA, BLOB, and CLOB columns, and you can change only the length. That is, you can change the data type from VARCHAR(10) to VARCHAR(100), but not from VARCHAR(10) to CLOB(100).
F383	Set column not null clause	Yes	None
F391	Long identifiers	Yes	None
F402	Named column joins for LOBs, arrays, and multisets	Yes	None
F431	Read-only scrollable cursors	Yes*	Through JDBC (only insensitive cursors)
F491	Constraint management	Yes	None
F492	Optional table constraint enforcement	Yes	None
F531	Temporary tables	Partial	Global tables (DECLARE GLOBAL TEMPORARY TABLE statement)
F591	Derived tables	Yes	None
F641	Row and table constructors	Yes	None
F690	Collation support	Partial	Users can create a database with territory-based collation
F701	Referential update actions	Partial	None
F721	Deferrable constraints	Partial	Deferrable NOT NULL constraints are not supported
F763	CURRENT_SCHEMA	Partial	Non-standard syntax (CURRENT SCHEMA instead of CURRENT_SCHEMA), and it is not allowed in a DEFAULT clause
F781	Self-referencing operations	Yes	None
F791	Insensitive cursors	Yes*	Through JDBC
F801	Full set function	Partial	DISTINCT in more than one aggregate function will not work, but SELECT

Feature ID	Feature Name	SQL:2011 Optional	Note
			DISTINCT with DISTINCT in one aggregate function will work
F850	Top-level <order by clause> in <query expression>	Yes	None
F851	<order by clause> in subqueries	Yes	None
F852	Top-level <order by clause> in views	Yes	None
F855	Nested <order by clause> in <query expression>	Yes	None
F856	Nested <fetch first clause> in <query expression>	Yes	None
F857	Top-level <fetch first clause> in <query expression>	Yes	None
F858	<fetch first clause> in subqueries	Yes	None
F859	Top-level <fetch first clause> in views	Yes	None
F860	Dynamic <fetch first row count> in <fetch first clause>	Yes	None
F861	Top-level <result offset clause> in <query expression>	Yes	None
F862	<result offset clause> in subqueries	Yes	None
F863	Nested <result offset clause> in <query expression>	Yes	None
F864	Top-level <result offset clause> in views	Yes	None
F865	Dynamic <offset row count> in <result offset clause>	Yes	None
J581	Output parameters	Yes	None
J621	External Java routines	Yes	None
J622	External Java types	Yes	None
T021	BINARY and VARBINARY data types	Yes*	Non-standard type names CHAR FOR BIT DATA and VARCHAR FOR BIT DATA instead of BINARY and VARBINARY
T031	BOOLEAN data type	Yes	None

Feature ID	Feature Name	SQL:2011 Optional	Note
T042	Extended LOB data type support	Partial	CAST and string value functions implemented. No comparison or ordering.
T071	BIGINT data type	Yes	None
T101	Enhanced nullability determination	Yes	None
T174	Identity columns	Yes*	MAXVALUE and CYCLE not supported. Deviation from standard: A comma (,) is required before INCREMENT.
T175	Generated columns	Yes	None
T176	Sequence generator support	Yes*	ALTER SEQUENCE not supported. Only one NEXT VALUE FOR clause per sequence in each statement
T191	Referential action RESTRICT	Yes	None
T212	Enhanced trigger capability	Yes	None
T271	Savepoints	Yes	None
T281	SELECT privilege with column granularity	Yes	None
T323	Explicit security for external routines	Yes	None
T326	Table functions	Partial	None
T331	Basic roles	Partial	None
T332	Extended roles	Partial	None
T351	Bracketed SQL comments (/*...*/ comments)	Yes	None
T431	Extended grouping capabilities	Partial	Partial support for GROUP BY ROLLUP
T441	ABS and MOD functions	Yes	None
T501	Enhanced EXISTS predicate	Yes	None
T591	UNIQUE constraints of possibly null columns	Yes	None
T611	Elementary OLAP operations	Partial	Partial support for ROW_NUMBER
X010	XML type	Yes	None
X016	Persistent XML values	Yes	None
X061	XMLParse: Character string input and DOCUMENT option	Partial	No support for the STRIP WHITESPACE option

Feature ID	Feature Name	SQL:2011 Optional	Note
X096	XMLExists	Partial	Support only for XPath queries, not full XQuery
X200	XMLQuery	Partial	Support only for XPath queries, not full XQuery
X202	XMLQuery: RETURNING SEQUENCE	Yes	None
X203	XMLQuery: passing a context item	Yes	None
X205	XMLQuery: EMPTY ON EMPTY option	Yes	None
X222	XML passing mechanism BY REF	Yes	None

SQL:2011 features not supported by Derby

Some mandatory and optional features in Part 2 of the SQL:2011 standard are not supported by Derby.

If a feature in another part of the standard is not listed in Derby support for SQL:2011 features, Derby does not support it.

Mandatory features

The following table lists the mandatory features in Part 2 of the SQL:2011 standard that are not supported by Derby and that are not listed in the tables in Derby support for SQL:2011 features.

Table 53. Mandatory SQL:2011 features not supported by Derby

Feature ID	Feature Name
E153	Updatable queries with subqueries
E182	Module language
F081	UNION and EXCEPT in views
F181	Multiple module support
F812	Basic flagging
S011	Distinct data types

Optional features

The following table lists the optional features in Part 2 of the SQL:2011 standard that are not supported by Derby and that are not listed in the tables in Derby support for SQL:2011 features.

Table 54. Optional SQL:2011 features not supported by Derby

Feature ID	Feature Name
F032	CASCADE drop behavior

Feature ID	Feature Name
F034	Extended REVOKE statement (F034-01 through F034-03)
F052	Intervals and datetime arithmetic
F053	OVERLAPS predicate
F121	Basic diagnostics management (F121-01, F121-02)
F171	Multiple schemas per user
F191	Referential delete actions
F222	INSERT statement: DEFAULT VALUES clause
F251	Domain support
F271	Compound character literals
F291	UNIQUE predicate
F301	CORRESPONDING in query expressions
F321	User authorization
F361	Subprogram support
F392	Unicode escapes in identifiers
F393	Unicode escapes in literals
F411	Time zone specification
F421	National character
F431	Read-only scrollable cursors (available through JDBC, but F431-01 through F431-06 are unsupported)
F441	Extended set function support
F442	Mixed column references in set functions
F451	Character set definition
F461	Named character sets
F521	Assertions
F555	Enhanced seconds precision
F561	Full value expressions
F571	Truth value tests
F611	Indicator data types
F651	Catalog name qualifiers
F661	Simple tables
F671	Subqueries in CHECK
F672	Retrospective check constraints
F692	Enhanced collation support
F693	SQL-session and client module collations
F695	Translation support
F711	ALTER domain

Feature ID	Feature Name
F731	INSERT column privileges
F741	Referential MATCH types
F751	View CHECK enhancements
F761	Session management
F771	Connection management
F813	Extended flagging
F821	Local table references
T051	Row types
T053	Explicit aliases for all-fields reference
T061	UCS support
T111	Updatable joins, unions, and columns
T121	WITH (excluding RECURSIVE) in query expression
T122	WITH (excluding RECURSIVE) in subquery
T131	Recursive query
T132	Recursive query in subquery
T141	SIMILAR predicate
T151	DISTINCT predicate
T152	DISTINCT predicate with negation
T171	LIKE clause in table definition
T172	AS subquery clause in table definition
T173	Extended LIKE clause in table definition
T201	Comparable data types for referential constraints
T231	Sensitive cursors
T241	START TRANSACTION statement
T251	SET TRANSACTION statement: LOCAL option
T261	Chained transactions
T272	Enhanced savepoint management
T301	Functional dependencies
T312	OVERLAY function
T324	Explicit security for SQL routines
T325	Qualified SQL parameter references
T432	Nested and concatenated GROUPING SETS
T433	Multiargument GROUPING function
T434	GROUP BY DISTINCT
T461	Symmetric BETWEEN predicate
T471	Result sets return value

Feature ID	Feature Name
T491	LATERAL derived table
T511	Transaction counts
T551	Optional key words for default syntax
T561	Holdable locators
T571	Array-returning external SQL-invoked functions
T572	Multiset-returning external SQL-invoked functions
T581	Regular expression substring function
T601	Local cursor references
T612	Advanced OLAP operations
T613	Sampling
T621	Enhanced numeric functions
T641	Multiple column assignment
T651	SQL-schema statements in SQL routines
T652	SQL-dynamic statements in SQL routines
T653	SQL-schema statements in external routines
T654	SQL-dynamic statements in external routines
T655	Cyclically dependent routines

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 system table

The SYSALIASES table describes the procedures, functions, user-defined types, and user-defined aggregates in the database.

The following table shows the contents of the SYSALIASES system table.

Table 55. SYSALIASES system table

Column Name	Type	Length	Nullable	Contents
ALIASID	CHAR	36	false	Unique identifier for the alias
ALIAS	VARCHAR	128	false	Alias (in the case of a user-defined type or user-defined aggregate, the name of the user-defined type or user-defined aggregate)
SCHEMAID	CHAR	36	true	Reserved for future use
JAVACLASSNAME	LONG VARCHAR	32,700	false	The Java class name
ALIASTYPE	CHAR	1	false	*'F'* (function), *'P'* (procedure), *'A'* (user-defined type), *'G'* (user-defined aggregate)
NAMESPACE	CHAR	1	false	*'F'* (function), *'P'* (procedure), *'A'* (user-defined type), *'G'* (user-defined aggregate)
SYSTEMALIAS	BOOLEAN	1	false	*true* (system supplied or built-in alias)

Column Name	Type	Length	Nullable	Contents
				false (alias created by a user)
ALIASINFO	*org.apache.derby.catalog.AliasInfo* This class is not part of the public API.	2,147,48	true	A Java interface that encapsulates the additional information that is specific to an alias
SPECIFICNAME	VARCHAR	128	false	System-generated identifier

SYSCHECKS system table

The SYSCHECKS table describes the check constraints within the current database.

The following table shows the contents of the SYSCHECKS system table.

Table 56. SYSCHECKS system table

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	false	Unique identifier for the constraint
CHECKDEFINITION	LONG VARCHAR	32,700	false	Text of check constraint definition
REFERENCEDCOLUMN	*org.apache.derby.catalog. Referenced* This class is not part of the public API.	2,147,48	false	Description of the columns referenced by the check constraint

SYSCOLPERMS system table

The SYSCOLPERMS table stores the column permissions that have been granted but not revoked.

All of the permissions for one (GRANTEE, TABLEID, TYPE, GRANTOR) combination are specified in a single row in the SYSCOLPERMS table. The keys for the SYSCOLPERMS table are:

- Primary key (GRANTEE, TABLEID, TYPE, GRANTOR)
- Unique key (COLPERMSID)
- Foreign key (TABLEID references SYS.SYSTABLES)

The following table shows the contents of the SYSCOLPERMS system table.

Table 57. SYSCOLPERMS system table

Column Name	Type	Length	Nullable	Contents
COLPERMSID	CHAR	36	false	Used by the dependency manager to track

Column Name	Type	Length	Nullable	Contents
				the dependency of a view, trigger, or constraint on the column level permissions
GRANTEE	VARCHAR	128	false	The authorization ID of the user or role to which the privilege was granted
GRANTOR	VARCHAR	128	false	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner
TABLEID	CHAR	36	false	The unique identifier for the table on which the permissions have been granted
TYPE	CHAR	1	false	If the privilege is non-grantable, the valid values are: *'s'* for SELECT *'u'* for UPDATE *'r'* for REFERENCES If the privilege is grantable, the valid values are: *'S'* for SELECT *'U'* for UPDATE *'R'* for REFERENCES
COLUMNS	*org.apache.derby.iapi.services.io.FormatableBitSet* This class is not part of the public API.	2,147,48	false	A list of columns to which the privilege applies

SYSCOLUMNS system table

The SYSCOLUMNS table describes the columns within all tables in the current database.

The following table shows the contents of the SYSCOLUMNS system table.

Table 58. SYSCOLUMNS system table

Column Name	Type	Length	Nullable	Contents
REFERENCEID	CHAR	36	false	Identifier for table (join with SYSTABLES.TABLEID)
COLUMNNAME	VARCHAR	128	false	Column or parameter name
COLUMNNUMBER	INT	10	false	The position of the column within the table
COLUMNDATATYPE	*org.apache.derby.catalog.TypeDescriptor* This class is not part of the public API.	2,147,48	false	System type that describes precision, length, scale, nullability, type name, and storage type of data. For a user-defined type, this column can hold a *TypeDescriptor* that refers to the appropriate type alias in SYS.SYSALIASES.
COLUMNDEFAULT	*java.io.Serializable*	2,147,48	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
AUTOINCREMENTVALU	BIGINT	20	true	What the next value for column will be, if the column is an identity column (Release 10.10 or earlier only)
AUTOINCREMENTSTAR	BIGINT	20	true	Initial value of column (if specified), if it is an identity column

Column Name	Type	Length	Nullable	Contents
AUTOINCREMENTINC	BIGINT	20	true	Amount column value is automatically incremented (if specified), if the column is an identity column

Note: The AUTOINCREMENTVALUE column has no meaning in a database that has been fully upgraded to Derby Release 10.11 or higher. At Release 10.11 or higher, use the SYSCS_UTIL.SYSCS_PEEK_AT_IDENTITY system function to observe the next value for an identity column.

SYSCONGLOMERATES system table

The SYSCONGLOMERATES table describes the conglomerates within the current database. A conglomerate is a unit of storage and is either a table or an index.

The following table shows the contents of the SYSCONGLOMERATES system table.

Table 59. SYSCONGLOMERATES system table

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)
CONGLOMERATENUMI	BIGINT	20	false	Conglomerate ID for the conglomerate (heap or index)
CONGLOMERATENAMI	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.IndexDescriptor* This class is not part of the public API.	2,147,48	true	System type describing the index
ISCONSTRAINT	BOOLEAN	1	true	Whether or not the conglomerate is a system-generated index enforcing a constraint

Column Name	Type	Length	Nullable	Contents
CONGLOMERATEID	CHAR	36	false	Unique identifier for the conglomerate

SYSCONSTRAINTS system table

The SYSCONSTRAINTS table describes the information common to all types of constraints within the current database (currently, this includes primary key, unique, foreign key, and check constraints).

The following table shows the contents of the SYSCONSTRAINTS system table.

Table 60. SYSCONSTRAINTS system table

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.SCHEM
STATE	CHAR	1	false	*'E'* (not deferrable initially immediate), *'i'* (deferrable initially immediate), or *'e'* (deferrable initially deferred)
REFERENCECOUNT	INTEGER	10	false	The count of the number of foreign key constraints that reference this constraint; this number can be greater than zero only for PRIMARY KEY and UNIQUE constraints

SYSDEPENDS system table

The SYSDEPENDS table stores the dependency relationships between persistent objects in the database.

Persistent objects can be dependents or providers. Dependents are objects that depend on other objects. Providers are objects that other objects depend on.

- Dependents are views, constraints, or triggers.
- Providers are tables, conglomerates, constraints, or privileges.

The following table shows the contents of the SYSDEPENDS system table.

Table 61. SYSDEPENDS system table

Column Name	Type	Length	Nullable	Contents
DEPENDENTID	CHAR	36	false	A unique identifier for the dependent
DEPENDENTFINDER	*org.apache.derby.catalog.DependableFinder* This class is not part of the public API.	2,147,48	false	A system type that describes the view, constraint, or trigger that is the dependent
PROVIDERID	CHAR	36	false	A unique identifier for the provider
PROVIDERFINDER	*org.apache.derby.catalog.DependableFinder* This class is not part of the public API.	2,147,48	false	A system type that describes the table, conglomerate, constraint, and privilege that is the provider

SYSFILES system table

The SYSFILES table describes jar files stored in the database.

The following table shows the contents of the SYSFILES system table.

Table 62. SYSFILES system table

Column Name	Type	Length	Nullable	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.SC
FILENAME	VARCHAR	128	false	SQL name of the jar file
GENERATIONID	BIGINT	20	false	Generation number for the file. When jar files are replaced, their generation

Column Name	Type	Length	Nullable	Contents
				identifiers are changed.

SYSFOREIGNKEYS system table

The SYSFOREIGNKEYS table 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.

The following table shows the contents of the SYSFOREIGNKEYS system table.

Table 63. SYSFOREIGNKEYS system table

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	false	Unique identifier for the foreign key constraint (join with SYSCONSTRAI
CONGLOMERATEID	CHAR	36	false	Unique identifier for index backing up the foreign key constraint (join with SYSCONGLOM CONGLOMERATEID)
KEYCONSTRAINTID	CHAR	36	false	Unique identifier for the primary key or unique constraint referenced by this foreign key (SYSKEYS.CONSTRAIN or SYSCONSTRAINTS.CONSTRAIN
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 system table

The SYSKEYS table 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.

The following table shows the contents of the SYSKEYS system table.

Table 64. SYSKEYS system table

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	false	Unique identifier for constraint
CONGLOMERATEID	CHAR	36	false	Unique identifier for backing index

SYSPERMS system table

The SYSPERMS table describes the USAGE permissions for sequence generators, user-defined types, and user-defined aggregates.

The following table shows the contents of the SYSPERMS system table.

Table 65. SYSPERMS system table

Column Name	Type	Length	Nullable	Contents
UUID	CHAR	36	false	The unique ID of the permission. This is the primary key.
OBJECTTYPE	VARCHAR	36	false	The kind of object receiving the permission. Valid values are *'SEQUENCE'*, *'TYPE'*, and *'DERBY AGGREGATE'*.
OBJECTID	CHAR	36	false	The UUID of the object receiving the permission. For sequence generators, the only valid values are SEQUENCEIDs in the SYS.SYSSEQUENCES table. For user-defined types and user-defined aggregates, the only valid values are ALIASIDs in the SYS.SYSALIASES table if the SYSALIASES row describes a user-defined type or user-defined aggregate.

Column Name	Type	Length	Nullable	Contents
PERMISSION	CHAR	36	false	The type of the permission. The only valid value is *'USAGE'*.
GRANTOR	VARCHAR	128	false	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner.
GRANTEE	VARCHAR	128	false	The authorization ID of the user or role to which the privilege was granted
ISGRANTABLE	CHAR	1	false	If the GRANTEE is the owner of the sequence generator, user-defined type, or user-defined aggregate, this value is *'Y'*. If the GRANTEE is not the owner of the sequence generator, user-defined type, or user-defined aggregate, this value is *'N'*.

SYSROLES system table

The SYSROLES table stores the roles in the database.

A row in the SYSROLES table represents one of the following:

- A role definition (the result of a CREATE ROLE statement)
- A role grant

The keys for the SYSROLES table are:

- Primary key (GRANTEE, ROLEID, GRANTOR)
- Unique key (UUID)

The following table shows the contents of the SYSROLES system table.

Table 66. SYSROLES system table

Column Name	Type	Length	Nullable	Contents
UUID	CHAR	36	false	A unique identifier for this role
ROLEID	VARCHAR	128	false	The role name, after conversion to case normal form
GRANTEE	VARCHAR	128	false	If the row represents a role grant, this is the authorization identifier of a user or role to which this role is granted. If the row represents a role definition, this is the database owner's user name.
GRANTOR	VARCHAR	128	false	This is the authorization identifier of the user that granted this role. If the row represents a role definition, this is the authorization identifier _SYSTEM. If the row represents a role grant, this is the database owner's user name (since only the database owner can create and grant roles).
WITHADMINOPTION	CHAR	1	false	A role definition is modelled as a grant from _SYSTEM to the database owner, so if the row represents a role definition, the value is always *'Y'*. This means that the creator (the database owner) is always allowed to grant the newly created role. Currently roles cannot be granted

Column Name	Type	Length	Nullable	Contents
				WITH ADMIN OPTION, so if the row represents a role grant, the value is always *'N'*.
ISDEF	CHAR	1	false	If the row represents a role definition, this value is *'Y'*. If the row represents a role grant, the value is *'N'*.

SYSROUTINEPERMS system table

The SYSROUTINEPERMS table stores the permissions that have been granted to routines.

Each routine EXECUTE permission is specified in a row in the SYSROUTINEPERMS table. The keys for the SYSROUTINEPERMS table are:

- Primary key (GRANTEE, ALIASID, GRANTOR)
- Unique key (ROUTINEPERMSID)
- Foreign key (ALIASID references SYS.SYSALIASES)

The following table shows the contents of the SYSROUTINEPERMS system table.

Table 67. SYSROUTINEPERMS system table

Column Name	Type	Length	Nullable	Contents
ROUTINEPERMSID	CHAR	36	false	Used by the dependency manager to track the dependency of a view, trigger, or constraint on the routine level permissions
GRANTEE	VARCHAR	128	false	The authorization ID of the user or role to which the privilege is granted
GRANTOR	VARCHAR	128	false	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner.
ALIASID	CHAR	36	false	The ID of the object of the required

Column Name	Type	Length	Nullable	Contents
				permission. If PERMTYPE='E', the ALIASID is a reference to the SYS.SYSALIASES table. Otherwise, the ALIASID is a reference to the SYS.SYSTABLES table.
GRANTOPTION	CHAR	1	false	Specifies if the GRANTEE is the owner of the routine. Valid values are 'Y' and 'N'.

SYSSCHEMAS system table

The SYSSCHEMAS table describes the schemas within the current database.

The following table shows the contents of the SYSSCHEMAS system table.

Table 68. SYSSCHEMAS system table

Column Name	Type	Length	Nullable	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

SYSSEQUENCES system table

The SYSSEQUENCES table describes the sequence generators in the database.

Note: Users should not directly query the SYSSEQUENCES table, because that will slow down the performance of sequence generators. Instead, users should call the SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE system function.

The following table shows the contents of the SYSSEQUENCES system table.

Table 69. SYSSEQUENCES system table

Column Name	Type	Length	Nullable	Contents
SEQUENCEID	CHAR	36	false	The ID of the sequence generator. This is the primary key.

Column Name	Type	Length	Nullable	Contents
SEQUENCENAME	VARCHAR	128	false	The name of the sequence generator. There is a unique index on (SCHEMAID, SEQUENCENAME)
SCHEMAID	CHAR	36	false	The ID of the schema that holds the sequence generator. There is a foreign key linking this column to SYSSCHEMAS.SCHEMAID.
SEQUENCEDATATYPE	*org.apache.derby.catalog.TypeDescriptor* This class is not part of the public API.	2,147,48	false	System type that describes the precision, length, scale, nullability, type name, and storage type of the data
CURRENTVALUE	BIGINT	20	true	The current value of the sequence generator. This is not the actual next value for the sequence generator. That value can be obtained by calling the system function SYSCS_UTIL.SY SYSSEQUENCES.CURRENTVALU holds the end of the range of values which have been preallocated in order to boost concurrency. The initial value of this column is STARTVALUE. This column is NULL only if the sequence generator is exhausted and cannot issue any more numbers.
STARTVALUE	BIGINT	20	false	The initial value of the sequence generator
MINIMUMVALUE	BIGINT	20	false	The minimum value of the

Column Name	Type	Length	Nullable	Contents
				sequence generator
MAXIMUMVALUE	BIGINT	20	false	The maximum value of the sequence generator
INCREMENT	BIGINT	20	false	The step size of the sequence generator
CYCLEOPTION	CHAR	1	false	If the sequence generator cycles, this value is *'Y'*. If the sequence generator does not cycle, this value is *'N'*.

SYSSTATEMENTS system table

The SYSSTATEMENTS table describes the prepared statements in the database.

The table contains one row per stored prepared statement.

The following table shows the contents of the SYSSTATEMENTS system table.

Table 70. SYSSTATEMENTS system table

Column Name	Type	Length	Nullable	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	1	false	Whether or not the statement is valid
TEXT	LONG VARCHAR	32,700	false	Text of the statement
LASTCOMPILED	TIMESTAMP	29	true	Time that the statement was compiled
COMPILATIONSCHEMA	CHAR	36	true	ID of the schema containing the statement
USINGTEXT	LONG VARCHAR	32,700	true	Text of the USING clause

Column Name	Type	Length	Nullable	Contents
				of the CREATE STATEMENT and ALTER STATEMENT statements

SYSSTATISTICS system table

The SYSSTATISTICS table describes the statistics within the current database.

The following table shows the contents of the SYSSTATISTICS system table.

Table 71. SYSSTATISTICS system table

Column Name	Type	Length	Nullable	Contents
STATID	CHAR	36	false	Unique identifier for the statistic
REFERENCEID	CHAR	36	false	The conglomerate for which the statistic was created (join with SYSCONGLOMER CONGLOMERATEID)
TABLEID	CHAR	36	false	The table for which the information is collected
CREATIONTIMESTAMP	TIMESTAMP	29	false	Time when this statistic was created or updated
TYPE	CHAR	1	false	Type of statistics
VALID	BOOLEAN	1	false	Whether the statistic is still valid
COLCOUNT	INTEGER	10	false	Number of columns in the statistic
STATISTICS	*org.apache.derby.catalog.Statistics* This class is not part of the public API.	2,147,48	false	Statistics information

SYSTABLEPERMS system table

The SYSTABLEPERMS table stores the table permissions that have been granted but not revoked.

All of the permissions for one (GRANTEE, TABLEID, GRANTOR) combination are specified in a single row in the SYSTABLEPERMS table. The keys for the SYSTABLEPERMS table are:

- Primary key (GRANTEE, TABLEID, GRANTOR)

- Unique key (TABLEPERMSID)
- Foreign key (TABLEID references SYS.SYSTABLES)

The following table shows the contents of the SYSTABLEPERMS system table.

Table 72. SYSTABLEPERMS system table

Column Name	Type	Length	Nullable	Contents
TABLEPERMSID	CHAR	36	false	Used by the dependency manager to track the dependency of a view, trigger, or constraint on the table level permissions
GRANTEE	VARCHAR	128	false	The authorization ID of the user or role to which the privilege is granted
GRANTOR	VARCHAR	128	false	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner
TABLEID	CHAR	36	false	The unique identifier for the table on which the permissions have been granted
SELECTPRIV	CHAR	1	false	Specifies if the SELECT permission is granted. The valid values are: *'y'* (non-grantable privilege) *'Y'* (grantable privilege) *'N'* (no privilege)
DELETEPRIV	CHAR	1	false	Specifies if the DELETE permission is granted. The valid values are: *'y'* (non-grantable privilege) *'Y'* (grantable privilege) *'N'* (no privilege)

Column Name	Type	Length	Nullable	Contents
INSERTPRIV	CHAR	1	False	Specifies if the INSERT permission is granted. The valid values are: *'y'* (non-grantable privilege) *'Y'* (grantable privilege) *'N'* (no privilege)
UPDATEPRIV	CHAR	1	False	Specifies if the UPDATE permission is granted. The valid values are: *'y'* (non-grantable privilege) *'Y'* (grantable privilege) *'N'* (no privilege)
REFERENCESPRIV	CHAR	1	false	Specifies if the REFERENCE permission is granted. The valid values are: *'y'* (non-grantable privilege) *'Y'* (grantable privilege) *'N'* (no privilege)
TRIGGERPRIV	CHAR	1	false	Specifies if the TRIGGER permission is granted. The valid values are: *'y'* (non-grantable privilege) *'Y'* (grantable privilege) *'N'* (no privilege)

SYSTABLES system table

The SYSTABLES table describes the tables and views within the current database.

The following table shows the contents of the SYSTABLES system table.

Table 73. SYSTABLES system table

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), *'A'* (synonym), or *'V'* (view)
SCHEMAID	CHAR	36	false	Schema ID for the table or view
LOCKGRANULARITY	CHAR	1	false	Lock granularity for the table: *'T'* (table level locking) or *'R'* (row level locking, the default)

SYSTRIGGERS system table

The SYSTRIGGERS table describes the database's triggers.

The following table shows the contents of the SYSTRIGGERS system table.

Table 74. SYSTRIGGERS system table

Column Name	Type	Length	Nullable	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.SC
CREATIONTIMESTAMP	TIMESTAMP	29	false	Time the trigger was created, for internal use to ensure correct execution order of triggers.
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

Column Name	Type	Length	Nullable	Contents
TABLEID	CHAR	36	false	ID of the table on which the trigger is defined
WHENSTMTID	CHAR	36	true	ID of the trigger's WHEN clause, if one is present
ACTIONSTMTID	CHAR	36	true	ID of the stored prepared statement for the *triggeredSQLStatement* (join with SYSSTATEMENTS.STMTI
REFERENCEDCOLUMN	*org.apache.derby. catalog. Referenced* This class is not part of the public API.	2,147,48	true	Descriptor of the columns to be updated, if this trigger is an update trigger (that is, if the EVENT column contains 'U')
TRIGGERDEFINITION	LONG VARCHAR	32,700	true	Text of the action SQL statement
REFERENCINGOLD	BOOLEAN	1	true	Whether or not the OLDREFERENC if non-null, refers to the OLD row or table
REFERENCINGNEW	BOOLEAN	1	true	Whether or not the NEWREFERENC if non-null, refers to the NEW row or table
OLDREFERENCINGNAI	VARCHAR	128	true	Pseudoname as set using the REFERENCING OLD AS clause
NEWREFERENCINGNA	VARCHAR	128	true	Pseudoname as set using the REFERENCING NEW AS clause
WHENCLAUSETEXT	LONG VARCHAR	32,700	true	Text of the trigger's WHEN clause, if one is present

Any SQL text that is part of a *triggeredSQLStatement* is compiled and stored in the SYSSTATEMENTS table. ACTIONSTMTID and WHENSTMTID are foreign keys that reference SYSSTATEMENTS.STMTID. The statements for a trigger are always in the same schema as the trigger.

SYSUSERS system table

The SYSUSERS table stores user credentials when NATIVE authentication is enabled.

When SQL authorization is enabled (as it is, for instance, when NATIVE authentication is on) only the database owner can SELECT from this table, and no one, not even the database owner, can SELECT the PASSWORD column.

The following table shows the contents of the SYSUSERS system table.

Table 75. SYSUSERS system table

Column Name	Type	Length	Nullable	Contents
USERNAME	VARCHAR	128	false	The user's name, the value of the `user` attribute on a connection URL.
HASHINGSCHEME	VARCHAR	32672	false	Describes how the password is hashed.
PASSWORD	VARCHAR	32672	false	The password after applying the HASHINGSCHEME.
LASTMODIFIED	TIMESTAMP	29	false	The time when the password was last updated.

SYSVIEWS system table

The SYSVIEWS table describes the view definitions within the current database.

The following table shows the contents of the SYSVIEWS system table.

Table 76. SYSVIEWS system table

Column Name	Type	Length	Nullable	Contents
TABLEID	CHAR	36	false	Unique identifier for the view (join with SYSTABLES.TAB
VIEWDEFINITION	LONG VARCHAR	32,700	false	Text of view definition
CHECKOPTION	CHAR	1	false	*'N'* (check option not supported yet)
COMPILATIONSCHEMA	CHAR	36	true	ID of the schema containing the view

XPLAIN style tables

Derby optionally creates database tables to hold statistics information captured using XPLAIN style. You can have zero, one, or many sets of these tables; each set of tables is stored in a separate schema. The schema which is used for capturing statement execution information is specified using the SYSCS_UTIL.SYSCS_SET_XPLAIN_SCHEMA system procedure

You can query these tables to analyze the behavior of statement execution.

All of the above system tables reside in the schema which you specified. Because this is not the default schema, qualify all queries accessing the system tables with the schema name.

You can create the schema and tables ahead of time if you wish, but usually it is easier to let Derby automatically create the schema and the tables for you. You can capture multiple sets of data into the same tables, or you can specify a different schema each time.

See "Working with RunTimeStatistics" in the *Tuning Derby* for additional information.

SYSXPLAIN_STATEMENTS system table

The SYSXPLAIN_STATEMENTS table captures information about statements which have been executed using RUNTIMESTATISTICS with XPLAIN style.

See "Working with RunTimeStatistics" in *Tuning Derby* for information on how to configure this.

Each row in this table describes a single statement which has been captured. Depending on the precise configuration of the RUNTIMESTATISTICS and XPLAIN features, there may be additional rows in the other XPLAIN system tables with additional information; the STMT_ID and TIMING_ID columns in this table are used to join against those tables.

Rows in this table are added automatically when Derby has been configured appropriately. The rows remain in the table until you delete them or drop the table.

The following table shows the contents of the SYSXPLAIN_STATEMENTS system table.

Table 77. SYSXPLAIN_STATEMENTS system table

Column Name	Type	Length	Nullable	Contents
STMT_ID	CHAR	36	false	A unique identifier for this particular captured statement.
STMT_NAME	VARCHAR	128	true	The name of the associated query or statement. This value is NULL if the user did not assign a name (by calling *java.sql.Statement.setCursorNam*
STMT_TYPE	CHAR	3	false	A code indicating what type of statement this is: *'S'* for SELECT, *'I'* for INSERT, *'U'*

Column Name	Type	Length	Nullable	Contents
				for UPDATE, *'D'* for DELETE, *'C'* for CALL, *'DDL'* for Data Definition (such as CREATE TABLE), *'SA'* for SELECT (Approximate), or blank, indicating the statement was a comment.
STMT_TEXT	VARCHAR	32,672	false	The text of the statement.
JVM_ID	VARCHAR	32,672	false	A code indicating what version of the JVM was running when this statement was captured. The code is a character that represents the release number plus one. For example, the code for Java SE 6 is *'7'*, and the code for Java SE 7 is *'8'*.
OS_IDENTIFIER	VARCHAR	32,672	false	Contains information about the operating system which was being used when this statement was captured.
XPLAIN_MODE	CHAR	1	true	A code indicating the XPLAIN mode which was in use when this statement was captured: *'F'* for FULL, or *'O'* for ONLY.
XPLAIN_TIME	TIMESTAMP	29	true	Contains the date and time when this statement was captured.
XPLAIN_THREAD_ID	VARCHAR	32,672	false	The JVM thread which was running when this statement was captured
TRANSACTION_ID	VARCHAR	32,672	false	An internal identifier for the transaction which was active when this statement was captured.
SESSION_ID	VARCHAR	32,672	false	An internal identifier for the session which was active when

Column Name	Type	Length	Nullable	Contents
				this statement was captured.
DATABASE_NAME	VARCHAR	128	false	Contains the name of the database which was being used when this statement was captured.
DRDA_ID	VARCHAR	32,672	true	In a network environment, this column contains an internal identifier for the network connection which was active when this statement was captured. In an embedded environment, this column is null.
TIMING_ID	CHAR	36	true	This field will be NULL unless SYSCS_UTIL.SYSCS_SET_STA has been called to enable statistics timing. If statistics timings are being captured, then this column will contain the ID of the row in SYSXPLAIN_STATE which records the statement timing for this statement.

SYSXPLAIN_STATEMENT_TIMINGS system table

The SYSXPLAIN_STATEMENT_TIMINGS table captures information about statement timings which occurred during statements that were executed using RUNTIMESTATISTICS with XPLAIN style.

See "Working with RunTimeStatistics" in *Tuning Derby* for information on how to configure this. Note in particular that you must call SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1) to enable timing information to be captured. Rows in this table are typically joined with rows in SYSXPLAIN_STATEMENTS and SYSXPLAIN_RESULTSETS during analysis. For example:

```
select s.stmt_text, st.execute_time
    from my_stats.sysxplain_statements s,
         my_stats.sysxplain_statement_timings st
    where s.timing_id = st.timing_id
    order by st.execute_time desc
```

Rows in this table are added automatically when Derby has been configured appropriately. The rows remain in the table until you delete them or drop the table.

The following table shows the contents of the SYSXPLAIN_STATEMENT_TIMINGS system table.

Table 78. SYSXPLAIN_STATEMENT_TIMINGS system table

Column Name	Type	Length	Nullable	Contents
TIMING_ID	CHAR	36	false	A unique identifier for this particular row. This column can be used to join with the TIMING_ID column in SYSXPLAIN_STATEMENTS to match statement timings with their corresponding statements.
PARSE_TIME	BIGINT	20	false	The time in milliseconds that Derby took to parse this statement.
BIND_TIME	BIGINT	20	false	The time in milliseconds that Derby took to bind this statement. Binding a statement is the process of resolving table and column references in the statement against the table and column definitions in the system catalogs.
OPTIMIZE_TIME	BIGINT	20	false	The time in milliseconds that Derby took to optimize this statement. During optimization, Derby considers the various possible execution plans that could be used for the statement, and chooses the one it thinks will be best.
GENERATE_TIME	BIGINT	20	false	The time in milliseconds that Derby took to generate code for this statement.
COMPILE_TIME	BIGINT	20	false	The time in milliseconds that Derby took to compile this statement. Overall statement time is divided into compile time and execute time, and the compile time is further sub-divided into parse, bind, optimize, and generate time.

Column Name	Type	Length	Nullable	Contents
EXECUTE_TIME	BIGINT	20	false	The time in milliseconds that Derby took to execute this statement.
BEGIN_COMP_TIME	TIMESTAMP	29	false	The time at which Derby began to compile this statement.
END_COMP_TIME	TIMESTAMP	29	false	The time at which Derby finished compiling this statement.
BEGIN_EXE_TIME	TIMESTAMP	29	false	The time at which Derby began to execute this statement.
END_EXE_TIME	TIMESTAMP	29	false	The time at which Derby finished executing this statement.

SYSXPLAIN_RESULTSETS system table

The SYSXPLAIN_RESULTSETS table captures information about each result set which is part of a statement that has been executed using RUNTIMESTATISTICS with XPLAIN style.

See "Working with RunTimeStatistics" in *Tuning Derby* for information on how to configure this.

Most statements have at least one result set associated with them, and some complex statements may have many result sets associated with them. Some statements, for example DDL statements such as CREATE TABLE, have no result sets associated with them.

Each row in this table describes a particular result set used by a particular statement. Rows in this table are typically joined with rows in SYSXPLAIN_STATEMENTS during analysis:

```
select st.stmt_text, rs.op_identifier
    from my_stats.sysxplain_statements st
    join my_stats.sysxplain_resultsets rs
        on st.stmt_id = rs.stmt_id
```

Rows in this table are added automatically when Derby has been configured appropriately. The rows remain in the table until you delete them or drop the table.

The following table shows the contents of the SYSXPLAIN_RESULTSETS system table.

Table 79. SYSXPLAIN_RESULTSETS system table

Column Name	Type	Length	Nullable	Contents
RS_ID	CHAR	36	false	A unique identifier for this particular row.
OP_IDENTIFIER	VARCHAR	32,672	false	A code indicating what type of result

Column Name	Type	Length	Nullable	Contents
				set these statistics are for. Common result set types include TABLESCAN, INDEXSCAN, and PROJECTION.
OP_DETAILS	VARCHAR	32,672	true	Additional string information which varies for each different type of result set. Interpreting this information currently requires reading the Derby source code to know what values are being displayed here.
NO_OPENS	INTEGER	10	true	Number of times this result set was opened during execution of the containing statement.
NO_INDEX_UPDATES	INTEGER	10	true	The number of index updates performed by this result set. This value is NULL for result sets used by queries, but may have a non-zero value for modification statements such as INSERT, UPDATE, or DELETE.
LOCK_MODE	CHAR	2	true	A code indicating the locking level that was used for this result set: *'EX'* for exclusive table-level locking, *'SH'* for share table-level locking, *'IX*' for exclusive row-level locking, or *'IS'* for share row-level locking.
LOCK_GRANULARITY	CHAR	1	true	A code indicating the locking granularity that was used for this result set: *'T'* for table-level locking, or *'R'* for row-level locking.
PARENT_RS_ID	CHAR	36	true	The result sets for a particular statement are arranged in a parent-child tree structure. The output

Column Name	Type	Length	Nullable	Contents
				rows from one result set are delivered as the input rows to its parent. This column stores the identifier of the parent result set. For the outermost result set in a particular statement, this column is NULL. Note that sometimes there are multiple result sets with the same parent result set (that is, some nodes have multiple children): for example, a UNION result set will have two child result sets, representing the two sets of rows which are UNIONed together.
EST_ROW_COUNT	DOUBLE	52	true	The optimizer's estimate of the total number of rows for this result set.
EST_COST	DOUBLE	52	true	The optimizer's estimated cost for this result set. The value indicates the number of milliseconds that the optimizer estimates it will take to process this result set.
AFFECTED_ROWS	INTEGER	10	true	This column is non-null only for INSERT, UPDATE, and DELETE result sets. For those result sets, this column holds the number of rows which were inserted, updated, or deleted, respectively.
DEFERRED_ROWS	CHAR	1	true	This column is only non-null for INSERT, UPDATE, and DELETE result sets. For those result sets, this column holds *'Y'* if the INSERT/UPDATE/DELETE is being performed using deferred change semantics, and

Column Name	Type	Length	Nullable	Contents
				holds *'N'* otherwise. Deferred change semantics are used when self-referencing is taking place.
INPUT_ROWS	INTEGER	10	true	This column is used for SORT, AGGREGATE, and GROUPBY result sets, and indicates the number of rows that were input to the result set, and thus were sorted by the sorter.
SEEN_ROWS	INTEGER	10	true	For join and set nodes, this is the number of rows seen by the "left" side of the processing. For aggregate, group, sort, normalize, materialize, and certain other nodes, this is the number of rows seen.
SEEN_ROWS_RIGHT	INTEGER	10	true	For join and set nodes, this is the number of rows seen by the "right" side of the processing. For example, in the statement `select country from countries union select country from countries where region = 'Africa'` the UNION result set has SEEN_ROWS = 6 and SEEN_ROWS_RIGHT = 19.
FILTERED_ROWS	INTEGER	10	true	This column holds the number of rows which were eliminated from the result set during processing.
RETURNED_ROWS	INTEGER	10	true	This column holds the number of rows which were returned by the result set to its caller. Generally speaking, the number of returned

Column Name	Type	Length	Nullable	Contents
				rows is the number of rows INPUT or SEEN, minus the number of rows FILTERED.
EMPTY_RIGHT_ROWS	INTEGER	10	true	This column is used for left outer joins, and, if not null, holds the number of empty rows which had to be constructed because no existing rows met the join criteria.

INDEX_KEY_OPT	CHAR	1	true	This column records when the Index Key Optimization is used. The Index Key Optimization is a special optimization which occurs when a query references the MAX or MIN value of a column which happens to have an index, and so the MIN or MAX computation can be performed by fetching the first or last, respectively, entry in the index, as in:
				`select max(country_iso_code)`
				`from countries`

SCAN_RS_ID	CHAR	36	true	If this resultset is one of the resultset types which performs a scan of a table or index, this column contains the id value which identifies the particular row in SYSXPLAIN_SCAN_ that describes the statistics related to the scan behavior.
SORT_RS_ID	CHAR	36	true	If this resultset is one of the resultset types which performs a sort of a table or index, this column contains the id value which identifies the particular row in SYSXPLAIN_SORT_ that describes the statistics related to the sort behavior. The most

Column Name	Type	Length	Nullable	Contents
				common situations which involve sorting of the data are when processing the ORDER BY and GROUP BY clauses.
STMT_ID	CHAR	36	false	This column will contain the ID value which identifies the particular statement for which this result set was executed. Note that there may be multiple result sets executed for a single statement, so a join between the SYSXPLAIN_STAT table and the SYSXPLAIN_RESULTSETS table may retrieve multiple rows.
TIMING_ID	CHAR	36	true	If statistics timings were not being captured, this column will have a NULL value. If statistics timings were being captured, this column will contain the id value which can be used as a foreign key to join with the SYSXPLAIN_RESULTSET_TIMINGS row which has the timing information for this resultset.

SYSXPLAIN_RESULTSET_TIMINGS system table

The SYSXPLAIN_RESULTSET_TIMINGS table captures timing information about result set accesses which occurred during statements that were executed using RUNTIMESTATISTICS with XPLAIN style.

See "Working with RunTimeStatistics" in *Tuning Derby* for information on how to configure this. Note that statistics timing must be configured by calling SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1). Each row in this table describes various timing information for this particular result set in this particular statement. Rows in this table are typically joined with rows in SYSXPLAIN_STATEMENTS and SYSXPLAIN_RESULTSETS during analysis. For example:

```
select rs.op_identifier, rst.execute_time
    from my_stats.sysxplain_resultsets rs,
         my_stats.sysxplain_resultset_timings rst
    where rs.stmt_id = ? and
          rs.timing_id = rst.timing_id
    order by rst.execute_time desc
```

Rows in this table are added automatically when Derby has been configured appropriately. The rows remain in the table until you delete them or drop the table.

The following table shows the contents of the SYSXPLAIN_RESULTSET_TIMINGS system table.

Table 80. SYSXPLAIN_RESULTSET_TIMINGS system table

Column Name	Type	Length	Nullable	Contents
TIMING_ID	CHAR	36	false	A unique ID for this particular row. This column can be used to join against the TIMING_ID column in the SYSXPLAIN_RESULTSETS table.
CONSTRUCTOR_TIME	BIGINT	20	true	The time it took to construct this instance of this result set, in milliseconds.
OPEN_TIME	BIGINT	20	true	The time it took to open this instance of this result set, in milliseconds. Note that if this result set was opened multiple times, this column is the sum of all the individual open times.
NEXT_TIME	BIGINT	20	true	The accumulated time for all the calls to fetch the next row from this result set, in milliseconds, for all the opens of this result set.
CLOSE_TIME	BIGINT	20	true	The time it took to close this instance of the result set, in milliseconds.
EXECUTE_TIME	BIGINT	20	true	The time for all operations performed by this result set, excluding the time taken by all the children result sets of this result set, in milliseconds.
AVG_NEXT_TIME_PER_R	BIGINT	20	true	If there was at least one row returned from this result set, then this value is the NEXT_TIME value divided by the number

Column Name	Type	Length	Nullable	Contents
				of rows returned from this result set, which thus is the average time, in milliseconds, that it took to retrieve a row from this result set.
PROJECTION_TIME	BIGINT	20	true	This value is NULL unless this result set is a PROJECTION result set, in which case this column contains the time, in milliseconds, that it took to perform projection of columns from the rows in this result set.
RESTRICTION_TIME	BIGINT	20	true	This value is NULL unless this result set is a PROJECTION result set, in which case this column contains the time, in milliseconds, that it took to perform restriction of rows from the rows in this result set.
TEMP_CONG_CREATE_T	BIGINT	20	true	For result sets which involve a materialization of a temporary intermediate result set, this value is the time it took to create the materialized result set, in milliseconds. This materialization may occur with hash joins where the number of rows in the intermediate result is too large to hold in memory.
TEMP_CONG_FETCH_TI	BIGINT	20	true	Similar to TEMP_CONG_CREATE_TIME this value is the time it took to retrieve rows from the materialized result set, in milliseconds.

SYSXPLAIN_SCAN_PROPS system table

The SYSXPLAIN_SCAN_PROPS table captures information about table/index accesses which occurred during statements that were executed using RUNTIMESTATISTICS with XPLAIN style.

See "Working with RunTimeStatistics" in *Tuning Derby* for information on how to configure this.

Each row in this table describes a single table/index scan for a particular result set used by a particular statement. Rows in this table are typically joined with rows in SYSXPLAIN_STATEMENTS and SYSXPLAIN_RESULTSETS during analysis:

```
select st.stmt_text, sp.no_visited_rows
    from my_stats.sysxplain_scan_props sp,
         my_stats.sysxplain_resultsets rs,
         my_stats.sysxplain_statements st
    where st.stmt_id = rs.stmt_id and
          rs.scan_rs_id = sp.scan_rs_id and
          rs.op_identifier = 'TABLESCAN' and
          sp.scan_object_name = 'COUNTRIES'
```

Rows in this table are added automatically when Derby has been configured appropriately. The rows remain in the table until you delete them or drop the table.

The following table shows the contents of the SYSXPLAIN_SCAN_PROPS system table.

Table 81. SYSXPLAIN_SCAN_PROPS system table

Column Name	Type	Length	Nullable	Contents
SCAN_RS_ID	CHAR	36	false	A unique identifer for this particular row. Referenced by the foreign key SCAN_RS_ID in SYSXPLAIN_RESULTS
SCAN_OBJECT_NAME	VARCHAR	128	true	The name of the object being scanned. If this is a scan of a table or index, the table name or index name appears here. If this is a scan of the internal index created for a constraint, the constraint name appears here. For complex join queries, the object being scanned may be an intermediate result, in which case a description such as *'Temporary HashTable'* appears.
SCAN_OBJECT_TYPE	CHAR	1	false	A code indicating the type of object being scanned. Codes include *'T'* for Table, *'I'* for Index, and *'C'* for Constraint.

Column Name	Type	Length	Nullable	Contents
SCAN_TYPE	CHAR	8	false	The type of scan being performed. Scan types include *'HEAP'*, *'BTREE'*, and *'SORT'*.
ISOLATION_LEVEL	CHAR	3	true	The isolation level being used for this scan. Isolation levels are identified by a code: *'RU'* for Read Uncommitted, *'RC'* for Read Committed, *'RR'* for Repeatable Read, and *'SE'* for Serializable.
NO_VISITED_PAGES	INTEGER	10	true	Number of database pages that this scan touched. For btree scans this number only includes the leaf pages visited.
NO_VISITED_ROWS	INTEGER	10	true	Number of database rows that were examined by this scan. This number includes all rows, including those rows marked deleted, those rows that don't meet qualification, and those rows which were returned by the scan.
NO_QUALIFIED_ROWS	INTEGER	10	true	Number of rows that satisfied the qualifiers for this scan.
NO_VISITED_DELETED_	INTEGER	10	true	Number of the database rows that were examined by this scan which were found to be rows that were marked deleted.
NO_FETCHED_COLUMN	INTEGER	10	true	Number of columns that were fetched from each qualifying row.
BITSET_OF_FETCHED_C	VARCHAR	32,672	true	Description of the columns which were fetched from each qualifying row.
BTREE_HEIGHT	INTEGER	10	true	For a scan of type BTREE, this column holds the height of the BTREE index.

Column Name	Type	Length	Nullable	Contents
				The typical height of a BTREE is 2-4; BTREE heights larger than this should only be seen with very large indexes. A tree with one page has a height of 1. Total number of pages visited in a scan of a BTREE should be (BTREE_HEIGHT - 1 + NO_VISITED_PAGES). For an extremely small BTREE, the btree height may be negative (-1). For other types of scans, this column is NULL.
FETCH_SIZE	INTEGER	10	true	The number of pages fetched at a time when the scan is retrieving pages from disk.
START_POSITION	VARCHAR	32,672	true	For index and constraint scans, a textual representation of the operator, if any, which was used to position the beginning of the index/constraint scan.
STOP_POSITION	VARCHAR	32,672	true	For index and constraint scans, a textual representation of the operator, if any, which was used to position the end of the index/constraint scan.
SCAN_QUALIFIERS	VARCHAR	32,672	true	If the query specified values which are to be used to limit the rows that are scanned, information about those values is captured in this column.
NEXT_QUALIFIERS	VARCHAR	32,672	true	If the query specified values which are to be used to limit the rows that are scanned, information about those values is captured in this column.

Column Name	Type	Length	Nullable	Contents
HASH_KEY_COLUMN_NU	VARCHAR	32,672	true	For hash joins, this column contains information about which column is being used to hash the rows that are joined.
HASH_TABLE_SIZE	INTEGER	10	true	For hash joins, this column contains information about the size of the hash table that will be used to hold the rows being joined. This hash table is an intermediate result, and will be discarded at the end of the query. If the hash table cannot fit in memory, it will automatically spill over to disk. Since the spillover to disk can have significant performance implications, this value can provide a clue that the hash table was unexpectedly too large to fit in memory.

SYSXPLAIN_SORT_PROPS system table

The SYSXPLAIN_SORT_PROPS table captures information about row sorting actions which occurred during statements that were executed using RUNTIMESTATISTICS with XPLAIN style.

See "Working with RunTimeStatistics" in *Tuning Derby* for information on how to configure this.

Rows in this table are typically joined with rows in SYSXPLAIN_STATEMENTS and SYSXPLAIN_RESULTSETS during analysis.

```
select s.stmt_text, rs.op_identifier,
       srt.no_input_rows, srt.no_output_rows
   from my_stats.sysxplain_sort_props srt,
        my_stats.sysxplain_resultsets rs,
        my_stats.sysxplain_statements s
   where rs.stmt_id = s.stmt_id and rs.sort_rs_id = srt.sort_rs_id
```

Rows in this table are added automatically when Derby has been configured appropriately. The rows remain in the table until you delete them or drop the table.

The following table shows the contents of the SYSXPLAIN_SORT_PROPS system table.

Table 82. SYSXPLAIN_SORT_PROPS system table

Column Name	Type	Length	Nullable	Contents
SORT_RS_ID	CHAR	36	false	A unique identifier for this row. Matches the corresponding value of SORT_RS_ID in the my_stats.SYSXPI row for the result set which required this sort to be performed.
SORT_TYPE	CHAR	2	true	A code indicating the type of sort that was performed. The code values include *'IN'* for an internal sort, and *'EX'* for an external sort. An internal sort is one which was entirely performed in-memory and did not overflow to any temporary files, while an external sort used one or more external files.
NO_INPUT_ROWS	INTEGER	10	true	Number of rows which were provided to the sorter.
NO_OUTPUT_ROWS	INTEGER	10	true	Number of rows which were returned by the sorter. Note that this may be fewer rows than were input, for example when the sorter is performing GROUP BY processing or is eliminating duplicates.
NO_MERGE_RUNS	INTEGER	10	true	Number of merge runs which were provided. This value will be NULL for an internal sort, but for an external sort it indicates how many times the intermediate sort files were merged together. External sorts are far more expensive than internal sorts, and each additional merge run that an external sort must perform adds considerably more to the overhead of the sort.

Column Name	Type	Length	Nullable	Contents
MERGE_RUN_DETAILS	VARCHAR	32,672	true	Additional information about the size of the merge runs. This value will be NULL for an internal sort.
ELIMINATE_DUPLICATES	CHAR	1	true	A code indicating whether or not this sort eliminated duplicates from the input. Valid values are *'Y'* and *'N'*. This column only applies for a sort which was NOT performing GROUP BY aggregation; for GROUP BY sorts this column is always NULL. See the DISTINCT_AGGREGATE column for the corresponding information for aggregating sorts.
IN_SORT_ORDER	CHAR	1	true	A code indicating whether or not the rows which were input to the sorter were already in sort order, which can happen if the rows were retrieved by using an index, or if an earlier phase of processing had already sorted the data. The code is *'Y'* if the rows are already in sorted order, and *'N'* otherwise.
DISTINCT_AGGREGATE	CHAR	1	true	A code indicating whether the aggregation process was computing distinct aggregates or not. Valid values are *'Y'* and *'N'*.

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 a *SQLException* with a *SQLState* starting with `0A`. The exception class is *java.sql.SQLFeatureNotSupportedException*. 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 Standard exceptions preceding those that are specific to Derby.

For information on processing *SQLExceptions*, see the *Derby Developer's Guide*.

SQL error messages and exceptions

The following tables list *SQLStates* for exceptions. Exceptions that begin with an *X* are specific to Derby.

Table 83. Class 01: Warning

SQLSTATE	Message Text
01001	An attempt to update or delete an already deleted row was made: No row was updated or deleted.
01003	Null values were eliminated from the argument of a column function.
01006	Privilege not revoked from user <*authorizationID*>.
01007	Role <*authorizationID*> not revoked from authentication id <*authorizationID*>.
01008	WITH ADMIN OPTION of role <*authorizationID*> not revoked from authentication id <*authorizationID*>.
01009	Generated column <*columnName*> dropped from table <*tableName*>.
0100E	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: <*indexName*>.
01505	The value <*dataValue*> 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.

SQLSTATE	Message Text
01J02	Scroll sensitive cursors 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 of failure.
01J06	ResultSet not updatable. Query does not qualify to generate an updatable ResultSet.
01J07	ResultSetHoldability restricted to ResultSet.CLOSE_CURSORS_AT_COMMIT for a global transaction.
01J08	Unable to open resultSet type <*resultSetType*>. ResultSet type <*resultSetType*> opened.
01J10	Scroll sensitive result sets are not supported by server; remapping to forward-only cursor
01J12	Unable to obtain message text from server. See the next exception. The stored procedure SYSIBM.SQLCAMESSAGE is not installed on the server. Please contact your database administrator.
01J13	Number of rows returned (<*number*>) is too large to fit in an integer; the value returned will be truncated.
01J14	SQL authorization is being used without first enabling authentication.
01J15	Your password will expire in <*remainingDays*> day(s). Please use the SYSCS_UTIL.SYSCS_MODIFY_PASSWORD procedure to change your password in database '<*databaseName*>'.
01J16	Your password is stale. To protect the database, you should update your password soon. Please use the SYSCS_UTIL.SYSCS_MODIFY_PASSWORD procedure to change your password in database '<*databaseName*>'.
01J17	You cannot encrypt, re-encrypt, or decrypt a database which is already booted. You must shutdown the database before attempting these operations.

Table 84. Class 07: Dynamic SQL Error

SQLSTATE	Message Text
07000	At least one parameter to the current statement is uninitialized.
07004	Parameter <*parameterName*> is an <*procedureName*> procedure parameter and must be registered with CallableStatement.registerOutParameter before execution.
07009	No input parameters.

Table 85. Class 08: Connection Exception

SQLSTATE	Message Text
08000	Connection closed by unknown interrupt.

SQLSTATE	Message Text
08001	A connection could not be established because the security token is larger than the maximum allowed by the network protocol.
08001	A connection could not be established because the user id has a length of zero or is larger than the maximum allowed by the network protocol.
08001	A connection could not be established because the password has a length of zero or is larger than the maximum allowed by the network protocol.
08001	A connection could not be established because the external name (EXTNAM) has a length of zero or is larger than the maximum allowed by the network protocol.
08001	A connection could not be established because the server name (SRVNAM) has a length of zero or is larger than the maximum allowed by the network protocol.
08001	Required Derby DataSource property <*propertyName*> not set.
08001	<*error*> : Error connecting to server <*serverName*> on port <*portNumber*> with message <*messageText*>.
08001	SocketException: '<*error*>'
08001	Unable to open stream on socket: '<*error*>'.
08001	User id length (<*number*>) is outside the range of 1 to <*number*>.
08001	Password length (<*length*>) is outside the range of 1 to <*number*>.
08001	User id can not be null.
08001	Password can not be null.
08001	A connection could not be established because the database name '<*databaseName*>' is larger than the maximum length allowed by the network protocol.
08003	No current connection.
08003	getConnection() is not valid on a closed PooledConnection.
08003	Lob method called after connection was closed
08003	The underlying physical connection is stale or closed.
08004	Connection refused : <*connectionName*>
08004	Connection authentication failure occurred. Reason: <*reasonText*>.
08004	The connection was refused because the database <*databaseName*> was not found.
08004	Database connection refused.
08004	User '<*authorizationID*>' cannot shut down database '<*databaseName*>'. Only the database owner can perform this operation.
08004	User '<*authorizationID*>' cannot (re)encrypt database '<*databaseName*>'. Only the database owner can perform this operation.
08004	User '<*authorizationID*>' cannot upgrade database '<*databaseName*>'. Only the database owner can perform this operation.
08004	Connection refused to database '<*databaseName*>' because it is in replication slave mode.

SQLSTATE	Message Text
08004	User '<*authorizationID*>' cannot issue a replication operation on database '<*databaseName*>'. Only the database owner can perform this operation.
08004	Missing permission for user '<*authorizationID*>' to shutdown system [<*exceptionMsg*>].
08004	Cannot check system permission to create database '<*databaseName*>' [<*exceptionMsg*>].
08004	Missing permission for user '<*authorizationID*>' to create database '<*databaseName*>' [<*exceptionMsg*>].
08004	Connection authentication failure occurred. Either the supplied credentials were invalid, or the database uses a password encryption scheme not compatible with the strong password substitution security mechanism. If this error started after upgrade, refer to the release note for DERBY-4483 for options.
08004	Username or password is null or 0 length.
08004	User '<*authorizationID*>' cannot decrypt database '<*databaseName*>'. Only the database owner can perform this operation.
08006	A network protocol error was encountered and the connection has been terminated: <*error*>
08006	An error occurred during connect reset and the connection has been terminated. See chained exceptions for details.
08006	SocketException: '<*error*>'
08006	A communications error has been detected: <*error*>.
08006	An error occurred during a deferred connect reset and the connection has been terminated. See chained exceptions for details.
08006	Insufficient data while reading from the network - expected a minimum of <*number*> bytes and received only <*number*> bytes. The connection has been terminated.
08006	Attempt to fully materialize lob data that is too large for the JVM. The connection has been terminated.
08006	org.apache.derby.jdbc.EmbeddedDriver is not registered with the JDBC driver manager
08006	Database '<*databaseName*>' shutdown.
08006	Database '<*databaseName*>' dropped.

Table 86. Class 0A: Feature not supported

SQLSTATE	Message Text
0A000	Feature not implemented: <*featureName*>.
0A000	The DRDA command <*commandName*> is not currently implemented. The connection has been terminated.
0A000	JDBC method is not yet implemented.
0A000	JDBC method <*methodName*> is not supported by the server. Please upgrade the server.

SQLSTATE	Message Text
0A000	resultSetHoldability property <*propertyName*> not supported
0A000	cancel() not supported by the server.
0A000	Security mechanism '<*mechanismName*>' is not supported.
0A000	The data type '<*datatypeName*>' is not supported.

Table 87. Class 0P: Invalid role specification

SQLSTATE	Message Text
0P000	Invalid role specification, role does not exist: '<*roleName*>'.
0P000	Invalid role specification, role not granted to current user or PUBLIC: '<*roleName*>'.

Table 88. Class 21: Cardinality Violation

SQLSTATE	Message Text
21000	Scalar subquery is only allowed to return a single row.
21000	A row in the target table qualifies for more than one DELETE or UPDATE action.

Table 89. Class 22: Data Exception

SQLSTATE	Message Text
22001	A truncation error was encountered trying to shrink <*dataType*> '<*dataValue*>' to length <*length*>.
22003	The resulting value is outside the range for the data type <*datatypeName*>.
22003	Year (<*year*>) exceeds the maximum '<*year*>'.
22003	Decimal may only be up to 31 digits.
22003	Overflow occurred during numeric data type conversion of '<*datatypeName*>' to <*datatypeName*>.
22003	The length (<*number*>) exceeds the maximum length (<*datatypeName*>) for the data type.
22005	Unable to convert a value of type '<*typeName*>' to type '<*typeName*>' : the encoding is not supported.
22005	The required character converter is not available.
22005	Unicode string cannot convert to Ebcdic string
22005	Unrecognized JDBC type. Type: <*typeName*>, columnCount: <*number*>, columnIndex: <*number*>.
22005	Invalid JDBC type for parameter <*parameterName*>.
22005	Unrecognized Java SQL type <*datatypeName*>.
22005	Unicode string cannot convert to UTF-8 string
22005	An attempt was made to get a data value of type '<*datatypeName*>' from a data value of type '<*datatypeName*>'.

SQLSTATE	Message Text
22007	The string representation of a date/time value is out of range.
22007	The syntax of the string representation of a date/time value is incorrect.
22008	'<*argument*>' is an invalid argument to the <*functionName*> function.
2200H	Sequence generator '<*schemaName*>.<*sequenceName*>' does not cycle. No more values can be obtained from this sequence generator.
2200L	Values assigned to XML columns must be well-formed Document nodes.
2200M	Invalid XML Document: <*parserError*>
2200V	Invalid context item for <*operatorName*> operator; context items must be well-formed Document nodes.
2200W	XQuery serialization error: Attempted to serialize one or more top-level Attribute nodes.
22011	The second or third argument of the SUBSTR function is out of range.
22011	The range specified for the substring with offset <*offset*> and len <*len*> is out of range for the String: <*str*>.
22012	Attempt to divide by zero.
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 '<*startIndex*>'. The string to search for is '<*searchString*>'. The string to search from is '<*fromString*>'.
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.
22020	Invalid trim string, '<*string*>'. The trim string must be exactly one character or NULL. It cannot be 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.
22028	The string exceeds the maximum length of <*number*>.
22501	An ESCAPE clause of NULL returns undefined results and is not allowed.
2201X	Invalid row count for OFFSET, must be >= 0.
2201W	Row count for FIRST/NEXT must be >= 1 and row count for LIMIT must be >= 0.
2201Z	NULL value not allowed for <*string*> argument.

Table 90. Class 23: Constraint Violation

SQLSTATE	Message Text
23502	Column '<*columnName*>' cannot accept a NULL value.
23503	<*statementType*> 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 '<*indexOrConstraintName*>' defined on '<*tableName*>'.
23506	The transaction was aborted because of a deferred constraint violation: Duplicate in unique or primary key constraint identified by '<*indexOrConstraintName*>' defined on '<*tableName*>'.
23507	Deferred constraint violation: Duplicate in unique or primary key constraint identified by '<*indexOrConstraintName*>' defined on '<*tableName*>'.
23513	The check constraint '<*constraintName*>' was violated while performing an INSERT or UPDATE on table '<*tableName*>'.
23514	The transaction was aborted because of a deferred constraint violation: Check constraint identified by '<*indexOrConstraintName*>' defined on <*tableName*> as '<*constraintText*>'.
23515	Deferred constraint violation: Check constraint identified by '<*indexOrConstraintName*>' defined on <*tableName*> as '<*constraintText*>'.
23516	The transaction was aborted because of a deferred constraint violation: Foreign key '<*indexOrConstraintName*>' defined on <*tableName*> referencing constraint '<*indexOrConstraintName*>' defined on <*tableName*>, key '<*keyName*>'.
23517	Deferred constraint violation: Foreign key constraint '<*indexOrConstraintName*>' defined on <*tableName*> referencing constraint '<*indexOrConstraintName*>' defined on <*tableName*>, key '<*keyName*>'.

Table 91. Class 24: Invalid Cursor State

SQLSTATE	Message Text
24000	Invalid cursor state - no current row.
24501	The identified cursor is not open.

Table 92. Class 25: Invalid Transaction State

SQLSTATE	Message Text
25001	Cannot close a connection while a transaction is still active.
25001	Invalid transaction state: active SQL transaction.
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.

Table 93. Class 28: Invalid Authorization Specification

SQLSTATE	Message Text
28502	The user name '<*authorizationID*>' is not valid.

Table 94. Class 2D: Invalid Transaction Termination

SQLSTATE	Message Text
2D521	setAutoCommit(true) invalid during global transaction.
2D521	Commit or Rollback invalid for application execution environment.

Table 95. Class 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.

Table 96. Class 39: External Routine Invocation Exception

SQLSTATE	Message Text
39004	A NULL value cannot be passed to a method which takes a parameter of primitive type '<*type*>'.

Table 97. Class 3B: Invalid SAVEPOINT

SQLSTATE	Message Text
3B001	SAVEPOINT, <*savepointName*> 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.

Table 98. Class 40: Transaction Rollback

SQLSTATE	Message Text
40001	A lock could not be obtained due to a deadlock, cycle of locks and waiters is: <*lockCycle*>. The selected victim is XID : <*transactionID*>.
40XC0	Dead statement. This may be caused by catching a transaction severity error inside this statement.

SQLSTATE	Message Text
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
40XL1	A lock could not be obtained within the time requested. The lockTable dump is: <*tableDump*>
40XL2	Self-deadlock.
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.
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 till it exits the quiescent state.
40XT7	Operation is not supported in an internal transaction.
40XT8	An internal error was identified by RawStore module. Internal state detail from the transaction is as follows: <*internalTransactionState*>

Table 99. Class 42: Syntax Error or Access Rule Violation

SQLSTATE	Message Text
42000	Syntax error or access rule violation; see additional errors for details.
42500	User '<*authorizationID*>' does not have <*permissionType*> permission on table '<*schemaNamet*>'.'<*tableName*>'.
42501	User '<*authorizationID*>' does not have <*permissionType*> permission on table '<*schemaNamet*>'.'<*tableName*>' for grant.
42502	User '<*authorizationID*>' does not have <*permissionType*> permission on column '<*columnName*>' of table '<*schemaName*>'.'<*tableName*>'.
42503	User '<*authorizationID*>' does not have <*permissionType*> permission on column '<*columnName*>' of table '<*schemaName*>'.'<*tableName*>' for grant.
42504	User '<*authorizationID*>' does not have <*permissionType*> permission on <*objectName*> '<*schemaName*>'.'<*tableName*>'.
42505	User '<*authorizationID*>' does not have <*permissionType*> permission on <*objectName*> '<*schemaName*>'.'<*tableName*>' for grant.
42506	User '<*authorizationID*>' is not the owner of <*objectName*> '<*schemaName*>'.'<*tableName*>'.
42507	User '<*authorizationID*>' can not perform the operation in schema '<*schemaName*>'.
42508	User '<*authorizationID*>' can not create schema '<*schemaName*>'. Only the database owner can issue this statement.

SQLSTATE	Message Text
42509	Specified grant or revoke operation is not allowed on object '<*objectName*>'.
4250A	User '<*authorizationID*>' does not have <*permissionName*> permission on object '<*schemaName*>'.'<*objectName*>'.
4250B	Invalid database authorization property '<*propertyName*>=<*propertyValue*>'.
4250C	User(s) '<*authorizationID*>' must not be in both read-only and full-access authorization lists.
4250D	Repeated user(s) '<*authorizationID*>' in access list '<*listName*>';
4250E	Internal Error: invalid <*authorizationID*> id in statement permission list.
4251A	Statement <*sqlText*> can only be issued by database owner.
4251B	PUBLIC is reserved and cannot be used as a user identifier or role name.
4251C	Role <*authorizationID*> cannot be granted to <*authorizationID*> because this would create a circularity.
4251D	Only the database owner can perform this operation.
4251E	No one can view the '<*tableName*>'.'<*columnName*>' column.
4251F	You cannot drop the credentials of the database owner.
4251G	Please set derby.authentication.builtin.algorithm to a valid message digest algorithm. The current authentication scheme is too weak to be used by NATIVE authentication.
4251H	Invalid NATIVE authentication specification. Please set derby.authentication.provider to a value of the form NATIVE:$credentialsDB or NATIVE:$credentialsDB:LOCAL (at the system level).
4251I	Authentication cannot be performed because the credentials database '<*databaseName*>' does not exist.
4251J	The value for the property '<*propertyName*>' is formatted badly.
4251K	The first credentials created must be those of the DBO.
4251L	The derby.authentication.provider property specifies '<*dbName*>' as the name of the credentials database. This is not a valid name for a database.
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.
42601	ALTER TABLE statement cannot add an IDENTITY column to a table.
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 '<*dataType*>' 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 '<*columnName*>' is invalid.
42622	The name '<*name*>' is too long. The maximum length is '<*number*>'.

SQLSTATE	Message Text
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 '<*sqlText*>' is invalid.
42815	The data type, length or value of arguments '<*dataType*>' and '<*dataType*>' is incompatible.
42818	Comparisons between '<*type*>' and '<*type*>' are not supported. Types must be comparable. String types must also have matching collation. If collation does not match, a possible solution is to cast operands to force them to the default collation (e.g. SELECT tablename FROM sys.systables WHERE CAST(tablename AS VARCHAR(128)) = 'T1')
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.
42831	'<*columnName*>' cannot be a column of a primary 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.
42878	The ORDER BY clause of a SELECT UNION statement only supports unqualified column references and column position numbers. Other expressions are not currently supported.
42879	The ORDER BY clause may not contain column '<*columnName*>', since the query specifies DISTINCT and that column does not appear in the query result.
4287A	The ORDER BY clause may not specify an expression, since the query specifies DISTINCT.
4287B	In this context, the ORDER BY clause may only specify a column number.
42884	No authorized routine named '<*routineName*>' of type '<*type*>' having compatible arguments was found.
42886	'<*parameterMode*>' parameter '<*parameterName*>' requires a parameter marker '?'.
42894	DEFAULT value or IDENTITY attribute value is not valid for column '<*columnName*>'.
428C1	Only one identity column is allowed in a table.

SQLSTATE	Message Text
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.
42909	The CREATE TABLE statement does not include a column list.
42915	Foreign Key '<*key*>' is invalid for the reason which follows: '<*detailedReason*>'.
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 '<*schemaNamet*>'.
4293A	A role cannot be created with the name '<*authorizationID*>', the SYS prefix is reserved.
42962	Long column type column or parameter '<*columnName*>' not permitted in declared global temporary tables or procedure definitions.
42995	The requested function does not apply to global temporary tables.
42X01	Syntax error: <*error*>.
42X02	<*parserException*>.
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/View '<*objectName*>' 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 '<*columnName*>' appears more than once in the CREATE TABLE statement.
42X13	Column name '<*columnName*>' appears more than once times in the column list of an INSERT statement.
42X14	'<*columnName*>' is not a column in table or VTI '<*tableOrVTIName*>'.
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 '<*joinOrder*>' is not valid as a joinOrder specification. Only the values FIXED and UNFIXED are valid.

SQLSTATE	Message Text
42X19	The WHERE, WHEN or HAVING clause or CHECK CONSTRAINT definition is a '<*dataType*>' expression. It must be a BOOLEAN expression.
42X19	The WHERE or HAVING clause or CHECK CONSTRAINT definition is an untyped parameter expression. It must be a BOOLEAN expression.
42X20	Syntax error; integer literal expected.
42X23	Cursor <*cursorName*> is not updatable.
42X24	Column <*columnName*> is referenced in the HAVING clause but is not in the GROUP BY list.
42X25	The '<*functionName*>' function is not allowed on the '<*1*>' type.
42X26	The class '<*className*>' for column '<*columnName*>' does not exist or is inaccessible. 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 '<*operatorName*>' to be ? parameters.
42X36	The '<*operatorName*>' operator is not allowed to take a ? parameter as an operand.
42X37	The unary '<*operatorName*>' 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 '<*propertyValue*>').
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 '<*number*>' in column specification.
42X45	<*dataType*> is an invalid type for argument number <*argumentNumber*> of <*operatorName*>.
42X46	There are multiple functions named '<*functionName*>'. Use the full signature or the specific name.
42X47	

SQLSTATE	Message Text
	There are multiple procedures named '<*procedureName*>'. Use the full signature or the specific name.
42X48	Value '<*number*>' is not a valid precision for <*dataType*>.
42X49	Value '<*invalidNumber*>' is not a valid integer literal.
42X50	No method was found that matched the method call <*className*>.<*methodName*>(<*parameterTypes*>), 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 '<*tableName*>'.
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 '<*viewName*>'.
42X57	The getColumnCount() for external virtual table '<*tableName*>' returned an invalid value '<*number*>'. 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.
42X59	The number of columns in each VALUES constructor must be the same.
42X60	Invalid value '<*insertMode*>' for insertMode property specified for table '<*tableName*>'.
42X61	Types '<*dataType*>' and '<*dataType*>' are not <*sqlOperator*> compatible.
42X62	'<*sqlText*>' 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 '<*propertyValue*>' 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*>'. It may be that the field exists, but it is not public, or that 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*>'.
42X70	

SQLSTATE	Message Text
	The number of columns in the table column list does not match the number of columns in the underlying query expression in the table definition for '<*tableName*>'.
42X71	Invalid data type '<*datatypeName*>' for column '<*columnName*>'.
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 <*className*>.<*methodName*>(<*parameterTypes*>) was ambiguous. (No single maximally specific method.)
42X74	Invalid CALL statement syntax.
42X75	No constructor was found with the signature <*className*>(<*parameterTypes*>). 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.
42X81	A query expression must return at least one column.
42X82	The USING clause returned more than one row. Only single-row ResultSets are permissible.
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 CASE expression must have a known type.
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*>'.
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	

SQLSTATE	Message Text
	Table '<*tableName*>' contains a constraint definition with column '<*columnName*>' which is not in the table.
42X94	<*sqlObjectType*> '<*objectName*>' does not exist.
42X96	The database class path contains an unknown jar file '<*fileName*>'.
42X97	Conflicting constraint characteristics for constraint.
42X98	Parameters are not allowed in a VIEW definition.
42X99	Parameters are not allowed in a TABLE definition.
42XA0	The generation clause for column '<*columnName*>' has data type '<*datatypeName*>', which cannot be assigned to the column's declared data type.
42XA1	The generation clause for column '<*columnName*>' contains an aggregate. This is not allowed.
42XA2	'<*sqlObjectName*>' cannot appear in a Generation Clause because it may return unreliable results.
42XA3	You may not override the value of generated column '<*columnName*>'.
42XA4	The generation clause for column '<*columnName*>' references other generated columns. This is not allowed.
42XA5	Routine '<*routineName*>' may issue SQL and therefore cannot appear in this context.
42XA6	'<*columnName*>' is a generated column. It cannot be part of a foreign key whose referential action for DELETE is SET NULL or SET DEFAULT, or whose referential action for UPDATE is CASCADE.
42XA7	'<*columnName*>' is a generated or identity column. You cannot change its default value.
42XA8	You cannot rename '<*columnName*>' because it is referenced by the generation clause of column '<*columnName*>'.
42XA9	Column '<*columnName*>' needs an explicit datatype. The datatype can be omitted only for columns with generation clauses.
42XAA	The NEW value of generated column '<*columnName*>' is mentioned in the BEFORE action of a trigger. This is not allowed.
42XAB	NOT NULL is allowed only if you explicitly declare a datatype.
42XAC	'INCREMENT BY' value can not be zero.
42XAE	'<*argName*>' value out of range of datatype '<*datatypeName*>'. Must be between '<*minValue*>' and '<*maxValue*>'.
42XAF	Invalid 'MINVALUE' value '<*minValue*>'. Must be smaller than 'MAXVALUE: <*maxValue*>'.
42XAG	Invalid 'START WITH' value '<*startValue*>'. Must be between '<*minValue*>' and '<*maxValue*>'.
42XAH	A NEXT VALUE FOR expression may not appear in many contexts, including WHERE, ON, HAVING, ORDER BY, DISTINCT, CASE, GENERATION, and AGGREGATE clauses as well as WINDOW functions and CHECK CONSTRAINTS.

SQLSTATE	Message Text
42XAI	The statement references the following sequence more than once: '<*sequenceName*>'.
42XAJ	The CREATE SEQUENCE statement has a redundant '<*clauseName*>' clause.
42XAK	The target table of a MERGE statement must be a base table.
42XAL	The source table of a MERGE statement must be a base table or table function.
42XAM	The source and target tables of a MERGE statement may not have the same correlation name.
42XAN	Constraint characteristics not allowed for NOT NULL.
42XAO	Subqueries are not allowed in the WHEN [NOT] MATCHED clauses of MERGE statements.
42XAP	Synonyms are not allowed as the source or target tables of MERGE statements.
42XAQ	The source and target tables of MERGE statements may not have derived column lists.
42XAR	The NEXT VALUE operator may not be used on a system-owned sequence generator.
42XBA	The schema, table or column does not exist or the column is not a string type.
42XBB	The table does not have a primary key.
42XBC	Type not supported by the Lucene optional tool: '<*typeName*>'
42XBD	Character not allowed in Derby identifiers used by the Lucene optional tool: '<*invalidCharacter*>'
42XBE	Lucene index does not exist.
42XBF	The schema doesn't exist or the current user isn't the DBO and doesn't own the schema.
42XBG	The luceneSupport tool has already been loaded.
42XBH	The luceneSupport tool has already been unloaded.
42XBI	Cannot drop '<*directoryName*>' because it is not a directory.
42XBJ	Cannot create a Lucene index involving a column named '<*columnName*>'. Try renaming the column by declaring a view.
42XBK	The current Lucene version '<*luceneVersion*>' cannot read an index created by Lucene version '<*indexVersion*>'.
42XBL	Lucene indexes cannot be created in an encrypted database and, conversely, a database containing a Lucene index cannot be encrypted.
42XBM	Argument '<*argumentName*>' may not be null.
42XBN	A field and a key have the same name: '<*fieldName*>'
42XBO	Duplicate or null field name: '<*fieldName*>'

SQLSTATE	Message Text
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.
42Y03	'<*statement*>' is not recognized as a procedure.
42Y03	'<*statement*>' is not recognized as a function.
42Y03	'<*statement*>' is a procedure but it is being used as a function.
42Y03	'<*statement*>' is a function but it is being called as a 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 <full java path>.<method name>.
42Y05	There is no Foreign Key named '<*key*>'.
42Y07	Schema '<*schemaNamet*>' 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.
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.
42Y26	Aggregates are not allowed in the GROUP BY list.
42Y26	Subqueries are not allowed in the GROUP BY list.
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 valid grouping expressions and valid aggregate expressions.
42Y32	

SQLSTATE	Message Text
	Aggregator class '<*className*>' for aggregate '<*aggregateName*>' on type <*type*> does not implement org.apache.derby.iapi.sql.execute.ExecAggregator.
42Y33	Aggregate <*aggregateName*> contains one or more aggregates.
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 '<*reference*>' is invalid, or is part of an invalid expression. For a SELECT list with a GROUP BY, the columns and expressions being selected may only contain valid grouping expressions and valid aggregate expressions.
42Y37	'<*javaTypeName*>' 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	'<*sqlObjectName*>' may not appear in a CHECK CONSTRAINT definition because it may return non-deterministic results.
42Y40	'<*columnName*>' appears multiple times in the UPDATE OF column list for trigger '<*triggerName*>'.
42Y41	'<*storedPreparedStatementName*>' cannot be directly invoked via EXECUTE STATEMENT because it is part of a trigger.
42Y42	Scale '<*scaleValue*>' is not a valid scale for a <*dataType*>.
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 '<*vtiName*>' 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	'<*sqlText*>' cannot be performed on '<*sqlObjectName*>' 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 '<*invalidNumber*>' for optimizer override '<*optimizerOverrideName*>'.

SQLSTATE	Message Text
42Y59	Invalid value, '<*invalidNumber*>', specified for hashInitialCapacity override. Value must be greater than 0.
42Y60	Invalid value, '<*invalidNumber*>', specified for hashLoadFactor override. Value must be greater than 0.0 and less than or equal to 1.0.
42Y61	Invalid value, '<*invalidNumber*>', specified for hashMaxCapacity override. Value must be greater than 0.
42Y62	'<*statement*>' is not allowed on '<*viewName*>' 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 '<*invalidNumber*>' 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 '<*schemaNamet*>' cannot be dropped.
42Y69	No valid execution plan was found for this statement. This may have many causes: 1) you specified a hash join strategy when hash join is not allowed (no optimizable equijoin), 2) you are attempting to join two external virtual tables, each of which references the other, and so the statement cannot be evaluated, 3) you have specified a plan shape which the optimizer would never consider.
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	'<*sqlText*>' 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	<*sqlKeyword*> triggers may only reference <*sqlKeyword*> 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.

SQLSTATE	Message Text
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*>'.
42Y98	'<*sqlObjectName*>' may not appear in this context because it may return unreliable results.
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 '<*tableName*>' because it has an enabled trigger (<*triggerName*>).
42Z15	Invalid type specified for column '<*columnName*>'. The type of a column may not be changed.
42Z16	Only columns of type VARCHAR, CLOB, and BLOB 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 '<*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 or unique constraint, which cannot have any nullable columns.
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 <*queryTreeNodeIdentifier*>.
42Z53	Internal Error: Type of activation to generate for node choice <*number*> is unknown.
42Z60	<*sqlText*> not allowed unless database property <*propertyName*> has value '<*propertyValue*>'.

SQLSTATE	Message Text
42Z70	Binding directly to an XML value is not allowed; try using XMLPARSE.
42Z71	XML values are not allowed in top-level result sets; try using XMLSERIALIZE.
42Z72	Missing SQL/XML keyword(s) '<*keywords*>' at line <*lineNumber*>, column <*columnNumber*>.
42Z73	Invalid target type for XMLSERIALIZE: '<*typeName*>'.
42Z74	XML feature not supported: '<*featureName*>'.
42Z75	XML query expression must be a string literal.
42Z76	Multiple XML context items are not allowed.
42Z77	Context item must have type 'XML'; '<*dataType*>' is not allowed.
42Z79	Unable to determine the parameter type for XMLPARSE; try using a CAST.
42Z90	Class '<*className*>' does not return an updatable ResultSet.
42Z91	SELECT
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
42Z9D	Procedures that modify SQL data are not allowed in BEFORE triggers.
42Z9D	'<*statement*>' statements are not allowed in '<*triggerName*>' triggers.
42Z9E	Constraint '<*constraintName*>' is not a <*constraintType*> constraint.
42Z9F	Too many indexes (<*index*>) on the table <*tableName*>. The limit is <*number*>.
42ZA0	Statement too complex. Try rewriting the query to remove complexity. Eliminating many duplicate expressions or breaking up the query and storing interim results in a temporary table can often help resolve this error.
42ZA1	Invalid SQL in Batch: '<*batch*>'.
42ZA2	Operand of LIKE predicate with type <*dataType*> and collation <*collationType*> is not compatable with LIKE pattern operand with type <*dataType*> and collation <*collationType*>.
42ZA3	The table will have collation type <*collationType*> which is different than the collation of the schema <*schemaName*> hence this operation is not supported .
42ZB1	Parameter style DERBY_JDBC_RESULT_SET is only allowed for table functions.
42ZB2	Table functions can only have parameter style DERBY_JDBC_RESULT_SET.
42ZB3	XML is not allowed as the datatype of a user-defined aggregate or of a column returned by a table function.

SQLSTATE	Message Text
42ZB4	'<*schemaName*>'.'<*functionName*>' does not identify a table function.
42ZB5	Class '<*className*>' implements VTICosting but does not provide a public, no-argument constructor.
42ZB6	A scalar value is expected, not a row set returned by a table function.
42ZB7	Illegal reference to column '<*columnName*>' by a table function or VTI.
42ZC0	Window '<*windowName*>' is not defined.
42ZC1	Only one window is supported.
42ZC2	Window function is illegal in this context: '<*clauseName*>' clause
42ZC3	A user defined aggregate may not have the name of an aggregate defined by the SQL Standard or the name of a builtin Derby function having one argument: '<*aggregateName*>'
42ZC4	User defined aggregate '<*schemaName*>'.'<*aggregateName*>' is bound to external class '<*className*>'. The parameter types of that class could not be resolved.
42ZC6	User defined aggregate '<*schemaName*>'.'<*aggregateName*>' was declared to have this input Java type: '<*javaDataType*>'. This does not extend the following actual bounding input Java type: '<*javaDataType*>'.
42ZC7	User defined aggregate '<*schemaName*>'.'<*aggregateName*>' was declared to have this return Java type: '<*javaDataType*>'. This does not extend the following actual bounding return Java type: '<*javaDataType*>'.
42ZC8	Implementing class '<*className*>' for user defined aggregate '<*schemaName*>'.'<*aggregateName*>' could not be instantiated or was malformed. Detailed message follows: <*detailedMessage*>
42ZC9	A varargs routine must have parameter style DERBY or DERBY_JDBC_RESULT_SET.
42ZCA	Parameter style DERBY is only allowed for varargs routines.
42ZCB	A varargs procedure may not return result sets.
42ZCC	Bad optimizer override. There are <*rowSourceCountInPlan*> row sources in the plan but there should be <*actualRowSourceCount*>.
42ZCD	Bad optimizer override. The plan is not a left-deep tree.
42ZCE	Bad optimizer override. Row sources have not been resolved.

Table 100. Class 57: DRDA Network Protocol: Execution Failure

SQLSTATE	Message Text
57017	There is no available conversion for the source code page, <*codePage*>, to the target code page, <*codePage*>. The connection has been terminated.

Table 101. Class 58: DRDA Network Protocol: Protocol Error

SQLSTATE	Message Text
58009	Network protocol exception: only one of the VCM, VCS length can be greater than 0. The connection has been terminated.

SQLSTATE	Message Text
58009	The connection was terminated because the encoding is not supported.
58009	Network protocol exception: actual code point, <*codePoint*>, does not match expected code point, <*codePoint*>. The connection has been terminated.
58009	Network protocol exception: DDM collection contains less than 4 bytes of data. The connection has been terminated.
58009	Network protocol exception: collection stack not empty at end of same id chain parse. The connection has been terminated.
58009	Network protocol exception: DSS length not 0 at end of same id chain parse. The connection has been terminated.
58009	Network protocol exception: DSS chained with same id at end of same id chain parse. The connection has been terminated.
58009	Network protocol exception: end of stream prematurely reached while reading InputStream, parameter #<*number*>. The connection has been terminated.
58009	Network protocol exception: invalid FDOCA LID. The connection has been terminated.
58009	Network protocol exception: SECTKN was not returned. The connection has been terminated.
58009	Network protocol exception: only one of NVCM, NVCS can be non-null. The connection has been terminated.
58009	Network protocol exception: SCLDTA length, <*length*>, is invalid for RDBNAM. The connection has been terminated.
58009	Network protocol exception: SCLDTA length, <*length*>, is invalid for RDBCOLID. The connection has been terminated.
58009	Network protocol exception: SCLDTA length, <*length*>, is invalid for PKGID. The connection has been terminated.
58009	Network protocol exception: PKGNAMCSN length, <*length*>, is invalid at SQLAM <*sqlApplicationManager*>. The connection has been terminated.
58010	A network protocol error was encountered. A connection could not be established because the manager <*managerCodePoint*> at level <*level*> is not supported by the server.
58014	The DDM command 0x<*distributedDataManagementCommand*> is not supported. The connection has been terminated.
58015	The DDM object 0x<*distributedDataManagementObject*> is not supported. The connection has been terminated.
58016	The DDM parameter 0x<*distributedDataManagementParameter*> is not supported. The connection has been terminated.
58017	The DDM parameter value 0x<*distributedDataManagementParameterValue*> is not supported. An input host variable may not be within the range the server supports. The connection has been terminated.

Table 102. Class X0: Execution exceptions

SQLSTATE	Message Text
X0A00	The select list mentions column '<*columnName*>' twice. This is not allowed in queries with GROUP BY or HAVING clauses. Try aliasing one of the conflicting columns to a unique name.
X0X02	Table '<*tableName*>' cannot be locked in '<*mode*>' mode.
X0X03	Invalid transaction state - held cursor requires same isolation level
X0X05	Table/View '<*tableName*>' does not exist.
X0X07	Cannot remove jar file '<*fileName*>' because it is on your derby.database.classpath '<*fileName*>'.
X0X0D	Invalid column array length '<*columnArrayLength*>'. To return generated keys, column array must be of length 1 and contain only the identity column.
X0X0E	Table '<*tableName*>' does not have an auto-generated column at column position '<*columnPosition*>'.
X0X0F	Table '<*tableName*>' does not have an auto-generated column named '<*columnName*>'.
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 '<*schemaNamet*>'.
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 '<*columnName*>' in index '<*indexName*>' and table '<*schemaName*>.<*tableName*>' do not match for row location <*rowLocation*>. The value in the index is '<*dataValue*>', while the value in the base table is '<*dataValue*>'. The full index key, including the row location, is '<*indexKey*>'. The suggested corrective action is to recreate the index.
X0X62	Inconsistency found between table '<*tableName*>' and index '<*indexName*>'. Error when trying to retrieve row location '<*rowLocation*>' from the table. The full index key, including the row location, is '<*indexKey*>'. The suggested corrective action is to recreate the index.
X0X63	Got IOException '<*exceptionText*>'.
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	<*sqlObjectType*> '<*sqlObjectName*>' does not exist.
X0X85	Index '<*indexName*>' was not created because '<*indexType*>' 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 '<*operationName*>' cannot be performed on object '<*objectName*>' because there is an open ResultSet dependent on that object.

SQLSTATE	Message Text
X0X99	Index '<*indexName*>' does not exist.
X0Y16	'<*sqlObjectName*>' is not a view. If it is a table, then use DROP TABLE instead.
X0Y23	Operation '<*operationName*>' cannot be performed on object '<*objectName*>' because VIEW '<*viewName*>' is dependent on that object.
X0Y24	Operation '<*operationName*>' cannot be performed on object '<*objectName*>' because STATEMENT '<*statement*>' is dependent on that object.
X0Y25	Operation '<*operationName*>' cannot be performed on object '<*sqlObjectName*>' because <*sqlObjectType*> '<*sqlObjectName*>' is dependent on that object.
X0Y26	Index '<*indexName*>' is required to be in the same schema as table '<*tableName*>'.
X0Y28	Index '<*indexName*>' cannot be created on system table '<*tableName*>'. Users cannot create indexes on system tables.
X0Y29	Operation '<*operationName*>' cannot be performed on object '<*objectName*>' because TABLE '<*tableName*>' is dependent on that object.
X0Y30	Operation '<*operationName*>' cannot be performed on object '<*objectName*>' because ROUTINE '<*routineName*>' is dependent on that object.
X0Y32	<*sqlObjectType*> '<*sqlObjectName*>' already exists in <*sqlObjectType*> '<*sqlObjectName*>'.
X0Y38	Cannot create index '<*indexName*>' 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 '<*constraintName*>' is invalid: the number of columns (<*number*>) does not match the number of columns in the referenced key (<*number*>).
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.
X0Y47	Constraint '<*constraintName*>' is invalid: the unique or primary key constraint on table '<*tableName*>' is deferrable and the referential action is CASCADE or SET NULL.
X0Y54	Schema '<*schemaNamet*>' 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 '<*indexName*>' on table

SQLSTATE	Message Text
	'<*schemaNamet*>.<*tableName*>' has <*number*> rows, but the base table has <*number*> rows. The suggested corrective action is to recreate the index.
X0Y56	'<*sqlText*>' is not allowed on the System table '<*tableName*>'.
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 <*rowCount*> 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.
X0Y63	The command on table '<*tableName*>' failed because null data was found in the primary key/index column(s). All columns in a primary 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	<*sqlObjectType*> '<*sqlObjectName*>' already exists.
X0Y69	DDL is not supported in trigger <*triggerName*>.
X0Y70	INSERT, UPDATE and DELETE are not permitted on table <*tableName*> because trigger <*triggerName*> is active.
X0Y71	Transaction manipulation such as SET ISOLATION is not permitted because trigger <*triggerName*> is active.
X0Y72	Bulk insert replace is not permitted on '<*tableName*>' because it has an enabled trigger (<*triggerName*>).
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.
X0Y78	<*javaInterfaceName*>.executeQuery() cannot be called because multiple result sets were returned. Use <*javaInterfaceName*>.execute() to obtain multiple results.
X0Y78	<*javaInterfaceName*>.executeQuery() was called but no result set was returned. Use <*javaInterfaceName*>.executeUpdate() for non-queries.
X0Y79	Statement.executeUpdate() cannot be called with a statement that returns a ResultSet.
X0Y80	ALTER table '<*tableName*>' failed. Null data found in column '<*columnName*>'.

SQLSTATE	Message Text
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.
X0Y84	Too much contention on sequence <*sequenceName*>. This is probably caused by an uncommitted scan of the SYS.SYSSEQUENCES catalog. Do not query this catalog directly. Instead, use the SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE function to view the current value of a sequence generator.
X0Y85	The Derby property '<*propertyName*>' identifies a class which cannot be instantiated: '<*className*>'. See the next exception for details.
X0Y85	The Derby property '<*propertyName*>' identifies a class which does not implement the org.apache.derby.catalog.SequencePreallocator interface.
X0Y86	Derby could not obtain the locks needed to release the unused, preallocated values for the sequence '<*schemaName*>'.'<*sequenceName*>'. As a result, unexpected gaps may appear in this sequence.
X0Y87	There is already an aggregate or function with one argument whose name is '<*schemaName*>'.'<*aggregateOrFunctionName*>'.
X0Y88	Unknown optional tool: '<*toolName*>'
X0Y88	The class '<*className*>' does not implement the org.apache.derby.iapi.sql.dictionary.OptionalTool interface.
X0Y89	Bad arguments passed to SYSCS_UTIL.SYSCS_REGISTER_TOOL(). Please consult the Reference Manual section which describes this system procedure.
X0Y90	Cannot create an instance of <*className*>. Maybe this class is not visible on the classpath. Maybe it doesn not have a 0-arg constructor.
X0Y91	Cannot change constraint mode of <*constraintName*>. It is not a deferrable constraint.
X0Y92	Cannot change the names of this table function's columns.

Table 103. Class XBCA: CacheService

SQLSTATE	Message Text
XBCA0	Cannot create new object with key <*keyValue*> in <*cacheName*> cache. The object already exists in the cache.

Table 104. Class 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*>.
XBCM4	Java class file format limit(s) exceeded: <*limitDescriptor*> in generated class <*className*>.

Table 105. Class XBCX: Cryptography

SQLSTATE	Message Text
XBCX0	Exception from Cryptography provider. See next exception for details.
XBCX1	Initializing cipher with illegal mode, must be either CipherFactory.ENCRYPT or CipherFactory.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 '<*paddingDirective*>' 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.
XBCXF	The class '<*className*>' representing the encryption provider cannot be found.
XBCXF	The class '<*className*>' does not implement the java.security.Provider interface.
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.
XBCXM	The length of the external encryption key must be an even number.
XBCXN	The external encryption key contains one or more illegal characters. Allowed characters for a hexadecimal number are 0-9, a-f and A-F.
XBCXO	Cannot encrypt, re-encrypt or decrypt the database when there is a global transaction in the prepared state.
XBCXQ	Cannot encrypt, re-encrypt or decrypt a read-only database.

SQLSTATE	Message Text
XBCXS	Cannot encrypt, re-encrypt or decrypt a database when it is in the log archive mode.
XBCXU	Encryption, re-encryption or decryption of a database failed: <*failureMessage*>
XBCXW	The message digest algorithm '<*algorithmName*>' is not supported by any of the available cryptography providers. Please install a cryptography provider that supports that algorithm, or specify another algorithm in the derby.authentication.builtin.algorithm property.

Table 106. Class 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 <*moduleName*>. Please ensure your classpath includes the correct Derby software.
XBM05	Startup failed due to missing product version information for <*productName*>.
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 <*subSubProtocol*> StorageFactory class <*className*>.
XBM0A	The database directory '<*directoryName*>' exists. However, it does not contain the expected '<*servicePropertiesName*>' file. Perhaps Derby was brought down in the middle of creating this database. You may want to delete this directory and try creating the database again.
XBM0B	Failed to edit/write service properties file: <*errorMessage*>
XBM0C	Missing privilege for operation '<*operation*>' on file '<*path*>': <*errorMessage*>
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 class path.
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 an instance of a class named '<*className*>'. This class name was the value of the derby.authentication.provider property and was expected to be the name of an application-supplied implementation of org.apache.derby.authentication.UserAuthenticator. The underlying problem was: <*detail*>
XBM0N	JDBC Driver registration with java.sql.DriverManager failed. See next exception for details.
XBM0P	Service provider is read-only. Operation not permitted.

SQLSTATE	Message Text
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*>.
XBM0W	An exception was thrown while creating an instance of class <*className*> registered for identifier <*identifierName*>.
XBM0X	Supplied locale description '<*localeID*>' 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.
XBM03	Supplied value '<*collationName*>' for collation attribute is invalid, expecting UCS_BASIC or TERRITORY_BASED.
XBM04	Collator support not available from the JVM for the database's locale '<*localeName*>'.
XBM0Y	Backup database directory <*directoryName*> not found. Please make sure that the specified backup path is right.
XBM0Z	Unable to copy file '<*fileName*>' to '<*fileName*>'. Please make sure that there is enough space and permissions are correct.

Table 107. Class XBD: Communication exceptions

SQLSTATE	Message Text
XBDA0	Login timeout exceeded.

Table 108. Class XCL: Execution exceptions

SQLSTATE	Message Text
XCL01	Result set 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 '<*datatypeName*>' into a data value of type '<*datatypeName*>'.
XCL13	The parameter position '<*parameterPosition*>' is out of range. The number of parameters for this prepared statement is '<*number*>'.

SQLSTATE	Message Text
XCL14	The column position '<*columnPosition*>' 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.
XCL18	Stream or LOB value cannot be retrieved more than once
XCL19	Missing row in table '<*tableName*>' for key '<*key*>'.
XCL20	Catalogs at version level '<*versionNumber*>' cannot be upgraded to version level '<*versionNumber*>'.
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.
XCL26	Parameter <*parameterName*> is not an output parameter.
XCL27	Return output parameters cannot be set.
XCL30	An IOException was thrown when reading a '<*dataType*>' 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 <*sqlText*>. (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 <*sqlText*>. (The referential constraint is self-referencing and the table is dependent in a relationship with a delete rule of CASCADE.)

SQLSTATE	Message Text
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.)
XCL47	Use of '<*sqlText*>' requires database to be upgraded from version <*versionNumber*> to version <*versionNumber*> or later.
XCL48	TRUNCATE TABLE is not permitted on '<*tableName*>' 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 '<*tableName*>' because it has an enabled DELETE trigger (<*triggerName*>).
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 '<*versionNumber*>'.
XCL51	The requested function can not reference tables in SESSION schema.
XCL52	The statement has been cancelled or timed out.

Table 109. Class XCW: Upgrade unsupported

SQLSTATE	Message Text
XCW00	Unsupported upgrade from '<*versionID*>' to '<*versionID*>'.

Table 110. Class XCX: Internal Utility Errors

SQLSTATE	Message Text
XCXA0	Invalid identifier.
XCXB0	Invalid database classpath: '<*classpath*>'.
XCXC0	Invalid id list.
XCXE0	You are trying to do an operation that uses the locale of the database, but the database does not have a locale.

Table 111. Class XCY: Derby Property Exceptions

SQLSTATE	Message Text
XCY00	Invalid value for property '<*propertyName*>'='<*propertyValue*>'.
XCY02	The requested property change is not supported '<*propertyName*>'='<*propertyValue*>'.
XCY03	Required property '<*propertyName*>' has not been set.
XCY04	Invalid syntax for optimizer overrides. The syntax should be -- DERBY-PROPERTIES propertyName = value [, propertyName = value]*
XCY05	Invalid setting of the derby.authentication.provider property. This property is already set to enable NATIVE authentication and cannot be changed.
XCY05	Invalid setting of the derby.authentication.provider property. To enable NATIVE authentication, use the SYSCS_UTIL.SYSCS_CREATE_USER procedure to store credentials for the database owner.

Table 112. Class XCZ: org.apache.derby.database.UserUtility

SQLSTATE	Message Text
XCZ00	Unknown permission '<*permissionName*>'.
XCZ01	Unknown user '<*authorizationID*>'.
XCZ02	Invalid parameter '<*propertyName*>'='<*propertyValue*>'.

Table 113. Class XD00: Dependency Manager

SQLSTATE	Message Text
XD004	Unable to store dependencies.

Table 114. Class XIE: Import/Export Exceptions

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 end of file at unexpected place on line <*lineNumber*>.
XIE0I	An IOException occurred while writing data to the file.

SQLSTATE	Message Text
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.
XIE0N	An invalid hexadecimal string '<*hexString*>' detected in the import file.
XIE0P	Lob data file <*fileName*> referenced in the import file not found.
XIE0Q	Lob data file name cannot be null.
XIE0R	Import error on line <*lineNumber*> of file <*fileName*>: <*details*>
XIE0S	The export operation was not performed, because the specified output file (<*fileName*>) already exists. Export processing will not overwrite an existing file, even if the process has permissions to write to that file, due to security concerns, and to avoid accidental file damage. Please either change the output file name in the export procedure arguments to specify a file which does not exist, or delete the existing file, then retry the export operation.
XIE0T	The export operation was not performed, because the specified large object auxiliary file (<*fileName*>) already exists. Export processing will not overwrite an existing file, even if the process has permissions to write to that file, due to security concerns, and to avoid accidental file damage. Please either change the large object auxiliary file name in the export procedure arguments to specify a file which does not exist, or delete the existing file, then retry the export operation.

Table 115. Class XJ: Connectivity Errors

SQLSTATE	Message Text
XJ004	Database '<*databaseName*>' not found.
XJ008	Cannot rollback or release a savepoint when in auto-commit mode.
XJ009	Use of CallableStatement required for stored procedure call or use of output parameters: <*sqlText*>
XJ010	Cannot issue SAVEPOINT when autoCommit is on.
XJ011	Cannot pass null for SAVEPOINT name.
XJ012	'<*interfaceName*>' already closed.
XJ013	No identifier 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.
XJ021	Type is not supported.
XJ022	Unable to set stream: '<*name*>'.

SQLSTATE	Message Text
XJ023	Input stream did not have exact amount of data as the requested length.
XJ025	Input stream cannot have negative length.
XJ028	The URL '<*urlValue*>' is not properly formed.
XJ030	Cannot set Autocommit On when in a nested connection.
XJ040	Failed to start database '<*databaseName*>' with class loader <*classLoader*>, see the next exception for details.
XJ041	Failed to create database '<*databaseName*>', see the next exception for details.
XJ042	'<*propertyValue*>' is not a valid value for property '<*propertyName*>'.
XJ044	'<*number*>' is an invalid scale.
XJ045	Invalid or (currently) unsupported isolation level, '<*levelName*>', passed to Connection.setTransactionIsolation(). 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.
XJ048	Conflicting boot attributes specified: <*attributes*>
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 thru XAResource interface.
XJ058	Cannot rollback a global transaction using the Connection, commit processing must go thru XAResource interface.
XJ059	Cannot close a connection while a global transaction is still active.
XJ05B	JDBC attribute '<*attributeName*>' has an invalid value '<*attributeValue*>', valid values are '<*attributeValues*>'.
XJ05C	Cannot set holdability ResultSet.HOLD_CURSORS_OVER_COMMIT for a global transaction.
XJ061	The '<*methodName*>' method is only allowed on scroll cursors.
XJ062	Invalid parameter value '<*invalidNumber*>' for ResultSet.setFetchSize(int rows).
XJ063	Invalid parameter value '<*invalidNumber*>' for Statement.setMaxRows(int maxRows). Parameter value must be >= 0.
XJ064	Invalid parameter value '<*invalidNumber*>' for setFetchDirection(int direction).
XJ065	Invalid parameter value '<*invalidNumber*>' for Statement.setFetchSize(int rows).
XJ066	Invalid parameter value '<*invalidNumber*>' for Statement.setMaxFieldSize(int max).

SQLSTATE	Message Text
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	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/CLOB's transaction may be committed, its connection closed or it has been freed.
XJ074	Invalid parameter value '<*invalidNumber*>' for Statement.setQueryTimeout(int seconds).
XJ076	The position argument '<*positionArgument*>' 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.
XJ078	Offset '<*invalidNumber*>' is either less than zero or is too large for the current BLOB/CLOB.
XJ079	The length specified '<*number*>' exceeds the size of the BLOB/CLOB.
XJ080	USING execute statement passed <*number*> parameters rather than <*number*>.
XJ081	Conflicting create/restore/recovery attributes specified.
XJ081	Invalid value '<*parameterValue*>' passed as parameter '<*parameterName*>' to method '<*methodName*>'
XJ085	Stream has already been read and end-of-file reached and cannot be re-used.
XJ086	This method cannot be invoked while the cursor is not on the insert row or if the concurrency of this ResultSet object is CONCUR_READ_ONLY.
XJ087	Sum of position('<*pos*>') and length('<*length*>') is greater than the size of the LOB plus one.
XJ088	Invalid operation: wasNull() called with no data retrieved.
XJ090	Invalid parameter: calendar is null.
XJ091	Invalid argument: parameter index <*indexNumber*> is not an OUT or INOUT parameter.
XJ093	Length of BLOB/CLOB, <*number*>, is too large. The length cannot exceed <*number*>.
XJ095	An attempt to execute a privileged action failed.
XJ097	Cannot rollback or release a savepoint that was not created by this connection.
XJ098	The auto-generated keys value <*keyValue*> is invalid
XJ099	The Reader/Stream object does not contain length characters
XJ100	

SQLSTATE	Message Text
	The scale supplied by the registerOutParameter method does not match with the setter method. Possible loss of precision!
XJ103	Table name can not be null
XJ104	Shared key length is invalid: <*invalidNumber*>.
XJ105	DES key has the wrong length, expected length <*number*>, got length <*number*>.
XJ106	No such padding
XJ107	Bad Padding
XJ108	Illegal Block Size
XJ110	Primary table name can not be null
XJ111	Foreign table name can not be null
XJ112	Security exception encountered, see next exception for details.
XJ113	Unable to open file <*fileName*> : <*error*>
XJ114	Invalid cursor name '<*cursorName*>'
XJ115	Unable to open resultSet with requested holdability <*invalidNumber*>.
XJ116	No more than <*number*> commands may be added to a single batch.
XJ117	Batching of queries not allowed by J2EE compliance.
XJ118	Query batch requested on a non-query statement.
XJ121	Invalid operation at current cursor position.
XJ122	No updateXXX methods were called on this row.
XJ123	This method must be called to update values in the current row or the insert row.
XJ124	Column not updatable.
XJ125	This method should only be called on ResultSet objects that are scrollable (type TYPE_SCROLL_INSENSITIVE).
XJ126	This method should not be called on sensitive dynamic cursors.
XJ128	Unable to unwrap for '<*interfaceName*>'
XJ200	Exceeded maximum number of sections <*number*>
XJ202	Invalid cursor name '<*cursorName*>'.
XJ203	Cursor name '<*cursorName*>' is already in use
XJ204	Unable to open result set with requested holdability <*holdValue*>.
XJ206	SQL text '<*sqlText*>' has no tokens.
XJ207	executeQuery method can not be used for update.
XJ208	Non-atomic batch failure. The batch was submitted, but at least one exception occurred on an individual member of the batch. Use getNextException() to retrieve the exceptions for specific batched elements.
XJ209	The required stored procedure is not installed on the server.
XJ210	The load module name for the stored procedure on the server is not found.

SQLSTATE	Message Text
XJ211	Non-recoverable chain-breaking exception occurred during batch processing. The batch is terminated non-atomically.
XJ212	Invalid attribute syntax: *<attributeSyntax>*
XJ213	The traceLevel connection property does not have a valid format for a number.
XJ214	An IO Error occurred when calling free() on a CLOB or BLOB.
XJ215	You cannot invoke other java.sql.Clob/java.sql.Blob methods after calling the free() method or after the Blob/Clob's transaction has been committed or rolled back.
XJ216	The length of this BLOB/CLOB is not available yet. When a BLOB or CLOB is accessed as a stream, the length is not available until the entire stream has been processed.
XJ217	The locator that was supplied for this CLOB/BLOB is invalid

Table 116. Class XK: Security Exceptions

SQLSTATE	Message Text
XK000	The security policy could not be reloaded: *<reason>*
XK001	Username not found in SYS.SYSUSERS.

Table 117. Class XN: Network Client Exceptions

SQLSTATE	Message Text
XN001	Connection reset is not allowed when inside a unit of work.
XN008	Query processing has been terminated due to an error on the server.
XN009	Error obtaining length of BLOB/CLOB object, exception follows.
XN010	Procedure name can not be null.
XN011	Procedure name length *<number>* is not within the valid range of 1 to *<number>*.
XN012	On *<operatingSystemName>* platforms, XA supports version *<versionNumber>* and above, this is version *<versionNumber>*
XN013	Invalid scroll orientation.
XN014	Encountered an Exception while reading from the stream specified by parameter #*<number>*. The remaining data expected by the server has been filled with 0x0. The Exception had this message: *<messageText>*.
XN015	Network protocol error: the specified size of the InputStream, parameter #*<number>*, is less than the actual InputStream length.
XN016	Encountered an Exception while trying to verify the length of the stream specified by parameter #*<number>*. The Exception had this message: *<messageText>*.
XN017	End of stream prematurely reached while reading the stream specified by parameter #*<number>*. The remaining data expected by the server has been filled with 0x0.

SQLSTATE	Message Text
XN018	Network protocol error: the specified size of the Reader, parameter #<*number*>, is less than the actual InputStream length.
XN019	Error executing a <*xaFunctionName*>, server returned <*xaError*>.
XN020	Error marshalling or unmarshalling a user defined type: <*messageDetail*>
XN021	An object of type <*sourceClassName*> cannot be cast to an object of type <*targetClassName*>.
XN022	A write chain that has transmitted data to the server cannot be reset until the request is finished and the chain terminated.
XN023	The stream specified by parameter #<*number*> is locator-based and requires a nested request on the same connection to be materialized. This is not supported.
XN024	Encountered an exception which terminated the connection, while reading from the stream specified by parameter #<*number*>. The Exception had this message: '<*messageText*>'.

Table 118. Class XRE: Replication Exceptions

SQLSTATE	Message Text
XRE00	This LogFactory module does not support replication.
XRE01	The log received from the master is corrupted.
XRE02	Master and Slave at different versions. Unable to proceed with Replication.
XRE03	Unexpected replication error. See derby.log for details.
XRE04	Could not establish a connection to the peer of the replicated database '<*dbname*>' on address '<*hostname*>:<*portname*>'.
XRE04	Connection lost for replicated database '<*dbname*>'.
XRE05	The log files on the master and slave are not in synch for replicated database '<*dbname*>'. The master log instant is <*masterfile*>:<*masteroffset*>, whereas the slave log instant is <*slavefile*>:<*slaveoffset*>. This is fatal for replication - replication will be stopped.
XRE06	The connection attempts to the replication slave for the database <*dbname*> exceeded the specified timeout period.
XRE07	Could not perform operation because the database is not in replication master mode.
XRE08	Replication slave mode started successfully for database '<*dbname*>'. Connection refused because the database is in replication slave mode.
XRE09	Cannot start replication slave mode for database '<*dbname*>'. The database has already been booted.
XRE10	Conflicting attributes specified. See reference manual for attributes allowed in combination with replication attribute '<*attribute*>'.
XRE11	Could not perform operation '<*command*>' because the database '<*dbname*>' has not been booted.
XRE12	Replication network protocol error for database '<*dbname*>'. Expected message type '<*expectedtype*>', but received type '<*receivedtype*>'.

SQLSTATE	Message Text
XRE20	Failover performed successfully for database '<*dbname*>', the database has been shutdown.
XRE21	Error occurred while performing failover for database '<*dbname*>', Failover attempt was aborted.
XRE22	Replication master has already been booted for database '<*dbname*>'
XRE23	Replication master cannot be started since unlogged operations are in progress, unfreeze to allow unlogged operations to complete and restart replication
XRE40	Could not perform operation because the database is not in replication slave mode.
XRE41	Replication operation 'failover' or 'stopSlave' refused on the slave database because the connection with the master is working. Issue the 'failover' or 'stopMaster' operation on the master database instead.
XRE42	Replicated database '<*dbname*>' shutdown.
XRE43	Unexpected error when trying to stop replication slave mode. To stop replication slave mode, use operation 'stopSlave' or 'failover'.

Table 119. Class XSAI: Store - access.protocol.interface

SQLSTATE	Message Text
XSAI2	The conglomerate (<*conglomerateNumber*>) requested does not exist.
XSAI3	Feature not implemented.

Table 120. Class XSAM: Store - AccessManager

SQLSTATE	Message Text
XSAM0	Exception encountered while trying to boot module for '<*interfaceName*>'.
XSAM2	There is no index or conglomerate with conglomerate id '<*conglomID*>' to drop.
XSAM3	There is no index or conglomerate with conglomerate 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 <*pageNumber*> in container <*containerName*> not found.

Table 121. Class XSAS: 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.

SQLSTATE	Message Text
XSAS3	The type of a row inserted into a sort does not match the sort's template.
XSAS6	Could not acquire resources for sort.

Table 122. Class XSAX: Store - access.protocol.XA statement

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.

Table 123. Class XSCB: 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.
XSCB4	A method on a btree open scan has been called prior to positioning the scan on the first row (i.e. no next() call has been made yet). The current state of the scan is (<*number*>).
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 = <*trueOrFalse*>, position key is null = <*trueOrFalse*>.
XSCB8	The btree conglomerate <*conglomerateNumber*> is closed.
XSCB9	Reserved for testing.

Table 124. Class XSCG0: Conglomerate

SQLSTATE	Message Text
XSCG0	Could not create a template.

Table 125. Class XSCH: Heap

SQLSTATE	Message Text
XSCH0	Could not create container.
XSCH1	Container <*containerName*> not found.
XSCH4	Conglomerate could not be created.

SQLSTATE	Message Text
XSCH5	In a base table there was a mismatch between the requested column number <*number*> and the maximum number of columns <*number*>.
XSCH6	The heap container with container id <*containerID*> is closed.
XSCH7	The scan is not positioned.
XSCH8	The feature is not implemented.

Table 126. Class XSDA: RawStore - Data.Generic statement

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 is not deleted
XSDA6	Column <*columnName*> 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
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 id.
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	Wrong page is gotten for record handle <*recordHandle*>.
XSDAL	Record handle <*recordHandle*> unexpectedly points to overflow page.
XSDAM	Exception during restore of a SQLData object of class <*className*>. The specified class cannot be instantiated.
XSDAN	Exception during restore of a SQLData object of class <*className*>. The specified class encountered an illegal access exception.
XSDAO	Internal error: page <*pageNumber*> attempted latched twice.

SQLSTATE	Message Text
XSDAP	Unexpected no space error while attempting to update a row on page <*pageId*>. Values of internal fields at time of error are as follows: slot = <*slot*>, recordId = <*recordId*>, newColumnList = <*columnList*>, nextColumn = <*nextColumn*>, mode = <*updateMode*>, nextPortionHandle = <*nextPortionHandle*>, page dump = <*pageDump*>.

Table 127. Class XSDB: RawStore - Data.Generic transaction

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*> : <*number*>
XSDB3	Container information cannot change once written: was <*number*>, now <*number*>
XSDB4	Page <*page*> is at version <*versionNumber*>, the log file contains change version <*versionNumber*>, either there are log records of this page missing, or this page did not get written out 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 <*databaseName*>.
XSDB7	Warning: Derby (instance <*derbyInstanceID*>) is attempting to boot the database <*databaseName*> even though Derby (instance <*derbyInstanceID*>) may still be active. Only one instance of Derby should boot a database at a time. Severe and non-recoverable corruption can result and may have already occurred.
XSDB8	Warning: Derby (instance <*derbyInstanceID*>) is attempting to boot the database <*databaseName*> even though Derby (instance <*derbyInstanceID*>) may still be active. Only one instance of Derby should boot a database at a time. Severe and non-recoverable corruption can result if 2 instances of Derby boot on the same database at the same time. The derby.database.forceDatabaseLock=true property has been 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, but it may be left behind in some shutdowns. It will be necessary to remove the file by hand in that case. It is important to verify that no other VM is accessing the database before deleting the db.lck file by hand.
XSDB9	Stream container <*containerName*> is corrupt.
XSDBA	Attempt to allocate object <*object*> failed.
XSDBB	Unknown page format at page <*page*>, page dump follows: <*hexDump*>
XSDBC	Write of container information to page 0 of container <*container*> failed. See nested error for more information.

Table 128. Class XSDF: RawStore - Data.Filesystem statement

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. The exception was: <*exceptionText*>.
XSDF3	Cannot create segment <*segmentName*>.
XSDF4	Exception during remove of file <*fileName*> for dropped container, file could not be removed <*detailedException*>.
XSDF6	Cannot find the allocation page <*page*>.
XSDF7	Newly created page failed to be latched <*pageKey*>
XSDF8	Cannot find page <*page*> to reuse.
XSDFB	Operation not supported by a read only database
XSDFD	Different page image read on 2 I/Os on Page <*page*>, first image has incorrect checksum, second image has correct checksum. Page images follows: <*hexDump*><*hexDump*>
XSDFF	The requested operation failed due to an unexpected exception.
XSDFH	Cannot backup the database, got an I/O Exception while writing to the backup container file <*fileName*>.
XSDFI	Error encountered while trying to write data to disk during database recovery. Check that the database disk is not full. If it is then delete unnecessary files, and retry connecting to the database. It is also possible that the file system is read only, or the disk has failed, or some other problem with the media. System encountered error while processing page <*page*>.
XSDFJ	Error encountered while trying to remove database file <*fileName*>, as part of encrypting or decrypting database files. Incorrect file or directory ownership or permissions could cause remove of the file to fail. Processes not controlled by Derby like backup or virus checkers could also be responsible.
XSDFK	Error encountered while trying to remove a jar file <*fileName*> stored in the database. Incorrect file or directory ownership or permissions could cause remove of the file to fail. Processes not controlled by Derby like backup or virus checkers could also be responsible.

Table 129. Class XSDG: RawStore - Data.Filesystem database

SQLSTATE	Message Text
XSDG0	Page <*page*> could not be read from disk.
XSDG1	Page <*page*> could not be written to disk, please check if the disk is full, or if a file system limit, such as a quota or a maximum file size, has been reached.
XSDG2	Invalid checksum on Page <*page*>, expected=<*number*>, on-disk version=<*number*>, page dump follows: <*hexDump*>
XSDG3	Meta-data for <*containerName*> could not be accessed to <*type*><*file*>
XSDG4	Unrecoverable internal error encountered while attempting to read low level metadata about the table or index. Please provide your

SQLSTATE	Message Text
	support organization with the following exception information: Failed: arraycopy of embryonic page byte[<*pageArrayLength*>] to container information byte[<*containerArrayLength*>]. Values of variables and constants: MAX_BORROWED_SPACE: <*maxBorrowedSpace*>, BORROWED_SPACE_OFFSET(<*borrowedSpaceOffset*>) + BORROWED_SPACE_LENG(<*borrowedSpaceLength*>) = <*arraycopySourcePosition*>; arraycopylength: <*maxBorrowableSpace*>; embryonic page <*hexdump*>.
XSDG5	Database is not in create mode when createFinished is called.
XSDG6	Data segment directory not found in <*directoryPath*> 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 '<*directoryName*>' during restore. Please make sure that there is enough space and permissions are correct.
XSDG9	Derby thread received an interrupt during a disk I/O operation, please check your application for the source of the interrupt.

Table 130. Class XSLA: RawStore - Log.Generic database exceptions

SQLSTATE	Message Text
XSLA0	Cannot flush the log file to disk <*filePath*>.
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	System will shutdown, got I/O Exception while accessing log file.
XSLA3	Log Corrupted, has invalid data in the log stream.
XSLA4	Error encountered when attempting to write the transaction recovery log. Most likely the disk holding the recovery log is full. If the disk is full, the only way to proceed is to free up space on the disk by either expanding it or deleting files not related to Derby. It is also possible that the file system and/or disk where the Derby transaction log resides is read-only. The error can also be encountered if the disk or file system has failed.
XSLA5	Cannot read log stream for some reason to rollback transaction <*transactionID*>.
XSLA6	Cannot recover the database.
XSLA7	Cannot redo operation <*operation*> in the log.
XSLA8	Cannot rollback transaction <*transactionID*>, trying to compensate <*undoableOperation*> operation with <*compensationOperation*>
XSLAA	The store has been marked for shutdown by an earlier exception.
XSLAB	Cannot find log file <*logfileName*>, please make sure your logDevice property is properly set with the correct path separator for your platform.
XSLAC	Database at <*directoryPath*> has a format incompatible with the current version of software. It may have been created by or upgraded by a later version.

SQLSTATE	Message Text
XSLAD	Log Record at instant <*logInstant*> in log file <*logfileName*> corrupted. Expected log record length <*length*>, real length <*length*>.
XSLAE	Control file at <*fileName*> 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 logging system has been marked to shut down due to an earlier problem and will not allow any more operations until the system shuts down and restarts.
XSLAK	Database has exceeded largest log file number <*number*>.
XSLAL	Log record size <*size*> exceeded the maximum allowable log file size <*size*>. Error encountered in log file <*logfileName*>, position <*position*>.
XSLAM	Cannot verify database format at <*directoryPath*> due to IOException: <*exceptionDetails*>
XSLAN	Database at <*directoryPath*> has an incompatible format with the current version of the software. The database was created by or upgraded by version <*versionNumber*>.
XSLAO	Recovery failed unexpected problem: <*detailedMessage*>.
XSLAP	Database at <*directoryPath*> is at version <*versionNumber*>. Beta databases cannot be upgraded.
XSLAQ	Cannot create log file at directory <*directoryName*>.
XSLAR	Unable to copy log file '<*logfileName*>' to '<*logFileName*>' during restore. Please make sure that there is enough space and permissions are correct.
XSLAS	Log directory <*directoryName*> not found in backup during restore. Please make sure that backup copy is the correct one and it is not corrupted.
XSLAT	The log directory '<*directoryName*>' exists. The directory might belong to another database. Check that the location specified for the logDevice attribute is correct.

Table 131. Class XSLB: 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.
XSLB5	Illegal truncationLWM instant <*truncationPoint*> for truncation point <*logInstant*>. Legal range is from <*logInstant*> to <*logInstant*>.
XSLB6	Trying to log a 0 or -ve length log Record.
XSLB8	Trying to reset a scan to <*logInstant*>, beyond its limit of <*logInstant*>.
XSLB9	Cannot issue any more change, log factory has been stopped.

Table 132. Class XSRS: RawStore - protocol.Interface statement

SQLSTATE	Message Text
XSRS0	Cannot freeze the database after it is already frozen.
XSRS1	Cannot backup the database to <*directoryPath*>, which is not a directory.
XSRS4	Error renaming file (during backup) from <*fileName*> to <*fileName*>.
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 on the fly.
XSRS9	Record <*recordName*> no longer exists
XSRSA	Cannot backup the database when unlogged operations are uncommitted. Please commit the transactions with backup blocking operations.
XSRSB	Backup cannot be performed in a transaction with uncommitted unlogged operations.
XSRSC	Cannot backup the database to <*directoryLocation*>, it is a database directory.

Table 133. Class XSTA2: XACT_TRANSACTION_ACTIVE

SQLSTATE	Message Text
XSTA2	A transaction was already active, when attempt was made to make another transaction active.

Table 134. Class XSTB: RawStore - Transactions.Basic system

SQLSTATE	Message Text
XSTB0	An exception was thrown during transaction abort.
XSTB2	Cannot log transaction changes, maybe trying to write to a read only database.
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.

Table 135. Class XXXXX: No SQLSTATE

SQLSTATE	Message Text
XXXXX	Normal database session close.

JDBC reference

This section provides reference information about Derby's implementation of the Java Database Connectivity (JDBC) API and documents the way it conforms to the JDBC APIs.

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 fully Java technology-enabled driver (Type 4 of the types described under "JDBC Architecture" in http://www.oracle.com/technetwork/java/overview-141217.html).

See the *Derby Developer's Guide* for task-oriented instructions on working with the driver.

This JDBC driver implements the standard JDBC interfaces. When invoked from an application running in the same Java Virtual Machine (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.

For information on the DataSource implementations provided by Derby, see DataSource classes.

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.

java.sql.Driver interface

The class that loads Derby's local JDBC driver is the class *org.apache.derby.jdbc.EmbeddedDriver*. The class that loads Derby's network client driver is the class *org.apache.derby.jdbc.ClientDriver*.

Usually, you will not need to create an instance of one of these classes, because the driver class is loaded and registered automatically when the *java.sql.DriverManager* class is initialized. That typically happens on the first call to a *DriverManager* method such as *DriverManager.getConnection*. This section describes a few exceptions to this rule.

With the embedded driver, if your application shuts down Derby or calls the *DriverManager.deregisterDriver* method, and you then want to reload the driver, call the *Class.forName().newInstance()* method to do so:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
```

You also need to call the *Class.forName* method in this way if you need to boot the Derby engine without actually connecting to it -- for example, if you want to start an embedded Network Server instance. See, for example, "Overview of the SimpleNetworkServer Sample program" in the *Derby Server and Administration Guide*.

The actual driver that gets registered in the *DriverManager* to handle the *jdbc:derby:* protocol is not the class *org.apache.derby.jdbc.EmbeddedDriver* or *org.apache.derby.jdbc.ClientDriver*; 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 a *getDriver* or *getConnection* method call.

java.sql.Driver.getPropertyInfo method

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 the *Derby Developer's Guide*.

java.sql.DriverManager.getConnection method

A Java application that uses 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* that contains 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. If you specify any attributes both on the connection URL and in a *Properties* object, the attributes on the connection URL override the attributes in the *Properties* object.

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.

The following section provides reference information on the connection URL syntax for applications with embedded databases. For information on the connection URL syntax for accessing the Network Server, see "Accessing the Network Server by using the

network client driver" in the *Derby Server and Administration Guide*. For more conceptual information, including examples, see "Connecting to databases" in 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 as follows.

```
jdbc:derby:[subsubprotocol:][databaseName][;attributes]*
```

This syntax has the following components.

- `jdbc:derby:`

  In JDBC terminology, `derby` is the *subprotocol* for connecting to a Derby database. The subprotocol is always `derby` and does not vary.
- *subsubprotocol:*

  subsubprotocol specifies where Derby looks for a database: in a directory, in memory, in a classpath, or in a jar file. *subsubprotocol* is one of the following:

  - `directory`: The default, which need not be specified explicitly. The database is in the file system, and the path name is either relative to the system directory or absolute.
  - `memory`: Databases exist only in main memory and are not written to disk. An in-memory database may be useful when there is no need to persist the database -- for example, in some testing situations. See "Using in-memory databases" in the *Derby Developer's Guide* for more information.
  - `classpath`: Databases are treated as read-only databases, relative to the `classpath` directory. 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.) See "Accessing databases from the classpath" in the *Derby Developer's Guide* for more information.
  - `jar`: Databases are treated as read-only databases. *DatabaseNames* might require a leading slash, because you specify them "relative" to the jar file. See "Accessing databases from a jar or zip file" in the *Derby Developer's Guide* for details.

    `jar` requires an additional element immediately before the database name:

    ```
    (pathToArchive)
    ```

    pathToArchive is the path name of the jar or zip file that holds the database.
- *databaseName*

  Specify the *databaseName* to connect to an existing database or a new one.

  The *databaseName* value can be either an absolute path name or a path name relative to the system directory. For example, `thisDB`, `databases/thisDB`, and `c:/databases/2014/january/thisDB` can all be valid values. See "Connecting to databases" and its subsections in the *Derby Developer's Guide*. The path separator in the connection URL is a forward slash (`/`), even in Windows path names. The *databaseName* value cannot contain a colon (`:`), except for the colon after the drive name in a Windows path name.
- *attributes*

  Specify zero 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 routine.

```
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.)

If you specify any attributes both on the connection URL and in a *Properties* object, the attributes on the connection URL override the attributes in the *Properties* object.

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.Connection interface

A Derby *Connection* object is not garbage-collected until all other JDBC objects created from that connection are 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.

The *Connection* interface extends *AutoCloseable* in JDK 7 and after. If you declare a connection in a *try*-with-resources statement and there is an error that the code does not catch, the JRE will attempt to close the connection automatically.

Note that a transaction-severity or higher exception causes Derby to abort an in-flight transaction. But a statement-severity exception does NOT roll back the transaction. Also note that Derby throws an exception if an attempt is made to close a connection with an in-flight transaction. Suppose now that a *Connection* is declared in a *try*-with-resources

statement, a transaction is in-flight, and an unhandled statement-severity error occurs inside the *try*-with-resources block. In this situation, Derby will raise a follow-on exception as the JRE exits the *try*-with-resources block. (For details on error severity levels, see *derby.stream.error.logSeverityLevel*.)

It is therefore always best to catch errors inside the *try*-with-resources block and to either roll back or commit, as appropriate, to ensure that there is no pending transaction when leaving the *try*-with-resources block. This action also improves application portability, since DBMSs differ in their semantics when trying to close a connection with a pending transaction.

The following table describes features of *Connection* methods that are specific to Derby.

Table 136. Implementation notes on *Connection* methods

Returns	Signature	Implementation Notes
PreparedStatement	*prepareStatement(String sql, int [] columnIndexes)*	Every column index in the array must correlate to an auto-increment column within the target table of the INSERT.
PreparedStatement	*prepareStatement(String sql, String [] columnNames)*	Every column name in the array must designate an auto-increment column within the target table of the INSERT.

See Autogenerated keys for details on the use of the two forms of the *Connection.prepareStatement* method shown in this table.

java.sql.Connection.setTransactionIsolation method

The transaction isolation levels *java.sql.Connection.TRANSACTION_SERIALIZABLE*, *java.sql.Connection.TRANSACTION_REPEATABLE_READ*, *java.sql.Connection.TRANSACTION_READ_COMMITTED*, and *java.sql.Connection.TRANSACTION_READ_UNCOMMITTED* are available from a Derby database.

TRANSACTION_READ_COMMITTED is the default isolation level.

Changing the current isolation for the connection with *setTransactionIsolation* commits the current transaction and begins a new transaction. For more details about transaction isolation, see "Locking, concurrency, and isolation" in the *Derby Developer's Guide*.

java.sql.Connection.setReadOnly method

The *java.sql.Connection.setReadOnly* method is supported.

See the section "Differences using the Connection.setReadOnly method" in the *Derby Server and Administration Guide* for more information.

java.sql.Connection.isReadOnly method

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. In addition, the following optional methods raise "Feature not supported" exceptions.

- *createArrayOf(java.lang.String, java.lang.Object[])*
- *createNClob( )*
- *createSQLXML( )*
- *createStruct(java.lang.String, java.lang.Object[])*
- *getTypeMap( )*
- *prepareStatement(java.lang.String, int[])*
- *prepareStatement(java.lang.String, java.lang.String[])*
- *setTypeMap(java.util.Map)*

java.sql.DatabaseMetaData interface

Derby implements the *java.sql.DatabaseMetaData* interface.

For methods that take a catalog parameter, always specify *null* for this parameter, since Derby does not support catalogs. Similarly, methods that return a catalog in the *ResultSet* will always return *null* in that column.

The Derby implementation of the *getColumns* method returns an empty *ResultSet* if the table specified in the third argument is a SYNONYM. The method reports "YES" as the value of IS_AUTOINCREMENT if a column is generated.

The Derby implementation of the *getResultSetHoldability* method returns *ResultSet.HOLD_CURSORS_OVER_COMMIT*.

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)*.

Columns in the ResultSets returned by getFunctionColumns and getProcedureColumns

Columns in the *ResultSet*s returned by *getFunctionColumns* and *getProcedureColumns* are as described by the API.

The following table shows Derby-specific details for some specific columns returned by *getFunctionColumns*.

Table 137. Columns in the *ResultSet* returned by *getFunctionColumns*

Column	Description
COLUMN_TYPE	A *short* indicating what the row describes. Is always *DatabaseMetaData.functionColumnIn* if it represents

Column	Description
	a parameter; *DatabaseMetaData.functionReturn* if it represents a return value; and *DatabaseMetaData.functionColumnResult* if it represents a column in a table function.
TYPE_NAME	Derby-specific name for the type.
NULLABLE	Always returns *DatabaseMetaData.functionNullable*.
REMARKS	A *String* which describes the Java type of the method parameter.
ORDINAL_POSITION	The ordinal position, starting from 1, for the input and output parameters for the function.
METHOD_ID	A Derby-specific column.
PARAMETER_ID	A Derby-specific column.

The following table shows Derby-specific details for some specific columns returned by *getProcedureColumns*.

Table 138. Columns in the *ResultSet* returned by *getProcedureColumns*

Column	Description
COLUMN_TYPE	A *short* indicating what the row describes. Is always *DatabaseMetaData.procedureColumnIn* if the parameter is (possibly implicitly) declared as an IN parameter; *DatabaseMetaData.procedureColumnInOut* if the parameter is declared as an INOUT parameter; and *DatabaseMetaData.procedureColumnOut* if the parameter is declared as an OUT parameter.
TYPE_NAME	Derby-specific name for the type.
NULLABLE	Always returns *DatabaseMetaData.procedureNullable*.
REMARKS	A *String* which describes the Java type of the method parameter.
ORDINAL_POSITION	The ordinal position, starting from 1, for the input and output parameters for the procedure.
METHOD_ID	A Derby-specific column.
PARAMETER_ID	A Derby-specific column.

java.sql.DatabaseMetaData.getBestRowIdentifier method

The *java.sql.DatabaseMetaData.getBestRowIdentifier* method looks for identifiers in a specific order. This order might not return a unique row.

The *java.sql.DatabaseMetaData.getBestRowIdentifier* method looks for identifiers in the following order:

- A primary key on the table
- A unique constraint or unique index on the table
- All of the columns in the table

Note: If the *java.sql.DatabaseMetaData.getBestRowIdentifier* method does not find a primary key, unique constraint, or unique index, the method must look for identifiers in all

of the columns in the table. When the method looks for identifiers this way, the method will always find a set of columns that identify a row. However, a unique row might not be identified if there are duplicate rows in the table.

java.sql.Statement interface

Derby does not implement the *setEscapeProcessing* method of *java.sql.Statement*. In addition, the *cancel* method raises a "Feature not supported" exception.

The following table describes features of *Statement* methods that are specific to Derby.

Table 139. Implementation notes on *Statement* methods

Returns	Signature	Implementation Notes
ResultSet	*getGeneratedKeys()*	If the user has indicated that auto-generated keys should be made available, this method returns the same results as a call to the IDENTITY_VAL_LOCAL function. Otherwise this method returns null.
boolean	*execute(String sql, int [] columnIndexes)*	Every column index in the array must correlate to an auto-increment column within the target table of the INSERT.
boolean	*execute(String sql, String [] columnNames)*	Every column name in the array must designate an auto-increment column within the target table of the INSERT.
int	*executeUpdate(String sql, int [] columnIndexes)*	Every column index in the array must correlate to an auto-increment column within the target table of the INSERT.
int	*executeUpdate(String sql, String [] columnNames)*	Every column name in the array must designate an auto-increment column within the target table of the INSERT.

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.

Autogenerated keys

JDBC's auto-generated 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 auto-increment feature, which allows users to create columns in tables for which the database system automatically assigns increasing integer values. Users can call the method *Statement.getGeneratedKeys* to retrieve the value of such a column. This method returns a *ResultSet* object with a column for the automatically generated key.

The Derby implementation of *Statement.getGeneratedKeys* returns meaningful results only if the last statement was a single-row insert or update statement. If it was a multi-row insert or update, *Statement.getGeneratedKeys* will return a result set with only a single row, even though it should return one row for each inserted or updated row.

Calling *ResultSet.getMetaData* on the *ResultSet* object returned by *getGeneratedKeys* produces a *ResultSetMetaData* object that is similar to that returned by IDENTITY_VAL_LOCAL.

Users can indicate that auto-generated columns should be made available for retrieval by passing one of the following values as a second argument to the *Connection.prepareStatement*, *Statement.execute*, or *Statement.executeUpdate* methods:

- A constant indicating that auto-generated keys should be made available. The specific constant to use is `Statement.RETURN_GENERATED_KEYS`.
- An array of the names of the columns in the inserted or updated row that should be made available. If any column name in the array does *not* designate an auto-increment column, Derby will throw an error with the Derby embedded driver. With the client driver, the one element column name is ignored currently and the value returned corresponds to the identity column. To ensure compatibility with future changes an application should ensure the column described is the identity column. If the column name corresponds to another column or a non-existent column then future changes may result in a value for a different column being returned or an exception being thrown.
- An array of the positions of the columns in the inserted or updated row that should be made available. If any column position in the array does *not* correlate to an auto-increment column, Derby will throw an error with the Derby embedded driver. With the client driver, the one element position array is ignored currently and the value returned corresponds to the identity column. To ensure compatibility with future changes an application should ensure the column described is the identity column. If the position corresponds to another column or a non-existent column then future changes may result in a value for a different column being returned or an exception being thrown.

Example

Assume that we have a table TABLE1 defined as follows:

```
CREATE TABLE TABLE1 (C11 int, C12 int GENERATED ALWAYS AS IDENTITY)
```

The following three code fragments will all do the same thing: that is, they will create a *ResultSet* that contains the value of `C12` that is inserted into TABLE1.

Code fragment 1:

```
Statement stmt = conn.createStatement();
stmt.execute(
    "INSERT INTO TABLE1 (C11) VALUES (1)",
    Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
```

Code fragment 2:

```
Statement stmt = conn.createStatement();
String [] colNames = new String [] { "C12" };
stmt.execute(
    "INSERT INTO TABLE1 (C11) VALUES (1)",
    colNames);
ResultSet rs = stmt.getGeneratedKeys();
```

Code fragment 3:

```
Statement stmt = conn.createStatement();
int [] colIndexes = new int [] { 2 };
stmt.execute(
    "INSERT INTO TABLE1 (C11) VALUES (1)",
    colIndexes);
ResultSet rs = stmt.getGeneratedKeys();
```

If there is no indication that auto-generated columns should be made available for retrieval, a call to *Statement.getGeneratedKeys* will return a null *ResultSet*.

java.sql.CallableStatement interface

Derby supports all methods of *CallableStatement* except *setBlob*, *getBlob*, *setClob*, and *getClob*.

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 the Java programming language. (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.

The following table shows the parameter array types and return types that correspond to JDBC types.

Table 140. INOUT parameter type correspondence

JDBC Type	Array Type for Method Parameter	Value and Return Type
BIGINT	*long[]*	*long*
BINARY	*byte[][]*	*byte[]*
BLOB	*java.sql.Blob[]*	*java.sql.Blob*
BOOLEAN	*boolean[]*	*boolean*
CLOB	*java.sql.Clob[]*	*java.sql.Clob*
DATE	*java.sql.Date[]*	*java.sql.Date*
DOUBLE	*double[]*	*double*
FLOAT	*double[]*	*double*
INTEGER	*int[]*	*int*
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	*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.PreparedStatement interface

Derby provides all the required JDBC 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

The *setXXXStream* methods request 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 run, the JDBC driver makes repeated calls to this input stream. Derby supports the following JDBC stream methods for *PreparedStatement* objects:

- *setBinaryStream*

  Use for streams that contain uninterpreted bytes
- *setAsciiStream*

  Use for streams that contain ASCII characters
- *setCharacterStream*

  Use for streams that contain characters

Note: Derby does not support the *setNCharacterStream* method or the deprecated *setUnicodeStream* method.
Note: These methods do not require that you specify the length of the stream. However, if you omit the length argument when the stream object is a LOB greater than a single page in size, performance will be impaired if you later retrieve the length of the LOB. If you are simply inserting or reading data, performance is unaffected.

The stream object passed to *setBinaryStream* and *setAsciiStream* can be either a standard Java stream object or the user's own subclass that implements the standard *java.io.InputStream* interface. The object passed to *setCharacterStream* must be a subclass of the abstract *java.io.Reader* class.

According to the JDBC standard, streams can be stored only in columns with the data types shown in the following table. The word "Preferred" indicates the preferred target data type for the type of stream. See Mapping of java.sql.Types to SQL types.

Table 141. Streamable JDBC data types

Column Data Type	Corresponding Java Type	AsciiStream	CharacterStream	BinaryStream
CLOB	*java.sql.Clob*	Yes	Yes	No
CHAR	None	Yes	Yes	No
VARCHAR	None	Yes	Yes	No
LONGVARCHAR	None	Preferred	Preferred	No
BINARY	None	Yes	Yes	Yes
BLOB	*java.sql.Blob*	Yes	Yes	Yes
VARBINARY	None	Yes	Yes	Yes
LONGVARBINARY	None	Yes	Yes	Preferred

Note: Streams cannot be stored in columns of the other built-in data types or columns of user-defined data types.

Example

The following code fragment shows how a user can store a streamed, ASCII-encoded *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();

// 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.ResultSet interface

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*.

JDBC does not define the sort of rounding to use for *ResultSet.getBigDecimal*. Derby uses *java.math.BigDecimal.ROUND_HALF_DOWN*.

The following table describes features of *ResultSet* methods that are specific to Derby.

Table 142. Implementation notes on *ResultSet* methods

Returns	Signature	Implementation Notes
void	*deleteRow()*	After the row is deleted, 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.
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,

Returns	Signature	Implementation Notes
		then *getConcurrency()* will return *ResultSet.CONCUR_READ_ONLY*.
boolean	*rowDeleted()*	For forward-only result sets this method always returns *false*, for scrollable result sets it returns *true* if the row has been deleted, via result set or positioned delete.
boolean	*rowInserted()*	Always returns *false*.
boolean	*rowUpdated()*	For forward-only result sets this method always returns *false*, for scrollable result sets it returns *true* if the row has been updated, via result set or positioned update.
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.

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 code fragment 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.Reader to get the data
        java.io.Reader ip = rs.getCharacterStream(1);

        // process the stream--this is just a generic way to
        // print the data
        char[] buff = new char[128];
        int size;
        while ((size = ip.read(buff)) != -1) {
            String chunk = new String(buff, 0, size);
            System.out.print(chunk);
        }
    }
    rs.close();
    s.close();
    conn.commit();
```

java.sql.ResultSetMetaData interface

Derby does not track the source or updatability of columns in *ResultSets*, and so always returns the constants shown for the methods listed in the following table.

Table 143. Implementation notes on *ResultSetMetadata* methods

Method Name	Value
isDefinitelyWritable	false
isReadOnly	false
isWritable	false

java.sql.SQLException class

Derby supplies values for the *getMessage*, *getSQLState*, and *getErrorCode* calls of *SQLException*.

In addition, Derby sometimes returns multiple *SQLExceptions* using the *nextException* chain. The first exception is always the most severe exception, with SQL Standard exceptions preceding those that are specific to Derby.

For information on processing *SQLExceptions*, see "Working with Derby SQLExceptions in an application" in the *Derby Developer's Guide*.

Exceptions raised by Derby will generally be one of the refined subclasses of *SQLException*. These refined exceptions are raised under the conditions described by their respective API documentation. The subclasses include the following. For a complete list, see the API documentation for your version of the Java SE platform.

- **java.sql.SQLClientInfoException**
- **java.sql.SQLDataException**
- **java.sql.SQLFeatureNotSupportedException**
- **java.sql.SQLIntegrityConstraintViolationException**
- **java.sql.SQLInvalidAuthorizationSpecException**
- **java.sql.SQLSyntaxErrorException**
- **java.sql.SQLTransactionRollbackException**
- **java.sql.SQLTransientConnectionException**

java.sql.SQLWarning class

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.

java.sql.SQLXML interface

The *java.sql.SQLXML* interface is the mapping for the SQL XML data type. However, Derby defines the XML data type and operators only in the SQL layer. There is no JDBC-side support for the XML data type and operators.

You cannot instantiate a *java.sql.SQLXML* object in Derby, or bind directly into an XML value or retrieve an XML value directly from a result set. You must bind and retrieve the

XML data as Java strings or character streams by explicitly specifying the XML operators, XMLPARSE and XMLSERIALIZE, as part of your SQL queries.

Additionally, Derby does not provide JDBC metadata support for the XML data type.

java.sql.Savepoint interface

The *Savepoint* interface contains 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.

Derby does not support savepoints within a trigger.

Derby does not release locks as part of the rollback to savepoint.

For more information on using savepoints, see the *Derby Developer's Guide*.

Mapping of java.sql.Types to SQL types

In Derby, the *java.sql.Types* are mapped to SQL data types.

The following table shows the mapping of *java.sql.Types* to SQL types.

Table 144. Mapping of *java.sql.Types* to SQL types

java.sql.Types	SQL Types
BIGINT	BIGINT
BINARY	CHAR FOR BIT DATA
BLOB	BLOB
BOOLEAN	BOOLEAN
CHAR	CHAR
CLOB	CLOB
DATE	DATE
DECIMAL	DECIMAL
DOUBLE	DOUBLE PRECISION
FLOAT	DOUBLE PRECISION[1]
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
SQLXML[2]	XML
TIME	TIME

java.sql.Types	SQL Types
TIMESTAMP	TIMESTAMP
VARBINARY	VARCHAR FOR BIT DATA
VARCHAR	VARCHAR

Notes:

1. 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.
2. SQLXML is only valid in JDBC 4.0 and later environments. SQLXML corresponds to the SQL type XML in Derby. However, Derby does not recognize the java.sql.Types.SQLXML data type and does not support any JDBC-side operations for the XML data type. Support for XML and the related operators is implemented only at the SQL layer. See XML data types for more.

Mapping of java.sql.Blob and java.sql.Clob interfaces

In the JDBC API, *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. BLOB and CLOB objects are collectively referred to as LOBs (large objects).

The Derby implementation of the *java.sql.Blob* and *java.sql.Clob* interfaces is LOCATOR-based, meaning that the implementation provides a logical pointer to a LOB rather than a complete copy of the object. Also, Derby does not materialize a LOB when you use the BLOB or CLOB data type. You can, however, call methods on a *java.sql.Blob* and *java.sql.Clob* object to materialize it (that is, to retrieve the entire object or parts of it).

You can access a LOB column only once within a row, by invoking a getter method on it.

To use the *java.sql.Blob* and *java.sql.Clob* features:

- Use the SQL BLOB type for columns which hold very large binary values.
- Use the SQL CLOB type for columns which hold very large string values.
- Use the *getBlob* and *getClob* methods of the *java.sql.ResultSet* interface to retrieve a LOB using its locator. You can then materialize all or part of the LOB by calling *Blob* and *Clob* methods. Alternatively, you can call the *ResultSet.getBytes* method to materialize a BLOB, and you can call the *ResultSet.getString* method to materialize a CLOB.

Casting between strings and BLOBs is not recommended because casting is platform- and database-dependent. See CAST function for more information.

As with other character datatypes, Derby treats CLOBs as unicode strings and writes them to disk using UTF8 encoding. With a Java database like Derby, you do not need to worry about character sets and codepages.

Restrictions on BLOB and CLOB objects (LOB-types)

- LOB-types cannot be compared for equality (=) and non-equality (!=, <>).
- LOB-typed values are not orderable, so <, <=, >, >= tests are not supported.
- LOB-types cannot be used in indices or as primary key columns.
- DISTINCT, GROUP BY, and ORDER BY clauses are also prohibited on LOB-types.
- LOB-types cannot be involved in implicit casting as other base-types.

Recommendation: 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.

The following table describes features of *java.sql.Blob* methods that are specific to Derby.

Table 145. Implementation notes on *java.sql.Blob* methods

Returns	Signature	Implementation Notes
byte[]	*getBytes(long pos, int length)*	Exceptions are raised if *pos* < 1, if *pos* is larger than the length of the *Blob*, or if *length* <= 0.
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*.

The following table describes features of *java.sql.Clob* methods that are specific to Derby.

Table 146. Implementation notes on *java.sql.Clob* methods

Returns	Signature	Implementation Notes
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	*position(Clob searchstr, long start)*	Exceptions are raised if *searchStr* == null or *start* < 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 on mapping of java.sql.Blob and java.sql.Clob interfaces

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*
- *getCharacterStream*

These methods share the same underlying stream; calling more than one of these methods on the same column 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.

Features supported on JDBC 4.1 and above

JDBC 4.1 added some functionality to the core API. This section documents the JDBC 4.1 features supported by Derby.

For information on features supported by all releases of JDBC 4, see JDBC reference. For information about features supported only by JDBC 4.2, see JDBC 4.2-only features.

Note: JDBC 4.1 features are present only in a JDK 7 or higher environment.

java.sql.Connection interface: JDBC 4.1 features

JDBC 4.1 adds new features to the *Connection* interface.

- **Aborting connections** - The *abort(Executor)* method aborts a running connection. Outstanding transactional work is rolled back, and the physical connection to the database is destroyed. When running under a Java *SecurityManager*, this method can be called only if *SQLPermission("callAbort")* has been granted both to the Derby JDBC driver (in *derby.jar* and *derbyclient.jar*) and to the application code that calls *Connection.abort()*. For security reasons, permission to execute this method should not be granted lightly. Do not grant this permission to application code unless you are certain that only superusers can invoke the code. For more information, see "Configuring Java security" in the *Derby Security Guide*.

JDBC 4.2-only features

JDBC 4.2 adds some functionality to the core API. This section documents the JDBC 4.2 features supported by Derby.

For information on features supported by all versions of JDBC 4, see JDBC reference. For information on features supported by both JDBC 4.1 and JDBC 4.2, see Features supported on JDBC 4.1 and above.

Note: JDBC 4.2 features are present only in a JDK 8 or higher environment.

JDBC support for Java SE 8 Compact Profiles

Derby provides support for Compact Profiles on the Java SE 8 platform by means of a group of JDBC DataSource classes.

These DataSource classes cannot be used in applications that use the Java Naming and Directory Interface (JNDI) API. Otherwise, they are just like the versions for the full Java SE platform.

Applications using Java SE 8 Compact Profile 2 *must* use these classes. Applications using Java SE 8 Compact Profile 3 can use the ordinary DataSource classes. (Compact Profile 1 does not provide any JDBC support, so Derby does not support it.)

The following DataSource classes are required for use with Java SE 8 Compact Profile 2:

- **org.apache.derby.jdbc.BasicEmbeddedDataSource40**
- **org.apache.derby.jdbc.BasicEmbeddedConnectionPoolDataSource40**
- **org.apache.derby.jdbc.BasicEmbeddedXADataSource40**
- **org.apache.derby.jdbc.BasicClientDataSource40**
- **org.apache.derby.jdbc.BasicClientConnectionPoolDataSource40**
- **org.apache.derby.jdbc.BasicClientXADataSource40**

java.sql.DatabaseMetaData interface: JDBC 4.2 features

Derby implements all of the new metadata methods added by JDBC 4.2.

The Derby implementation of the *getMaxLogicalLOBSize* method returns zero (0). For details on the meaningful limits on Derby's BLOB and CLOB datatypes, see BLOB data type and CLOB data type.

java.sql.SQLType interface

JDBC 4.2 introduces a new data type identifier, *java.sql.SQLType*, to help databases describe data types which do not appear in the ANSI/ISO SQL Standard. Databases which provide non-standard types can provide their own implementations of *SQLType*.

JDBC 4.2 also supplies its own implementation, *java.sql.JDBCType*, which provides an enum for each of the type identifiers in *java.sql.Types*.

Derby does not expose any datatypes which are not represented by *JDBCType* enums. Therefore, Derby does not need to provide its own implementation of *SQLType*.

Overloads with *SQLType* arguments have been added to a few interfaces, alongside the existing methods which take *int* type identifiers from *java.sql.Types*. The affected interfaces are as follows:

- *java.sql.CallableStatement*
- *java.sql.PreparedStatement*
- *java.sql.ResultSet*

With Derby, these methods raise a *java.sql.SQLFeatureNotSupportedException* (SQLState 0A000) if the caller passes in a bad *SQLType*, which can be either of the following:

- A *SQLType* from a foreign database; that is, a *SQLType* which is not one of the *JDBCType* enums.
- A *JDBCType* enum whose corresponding *int* type identifier (from *java.sql.Types*) is not supported by Derby. The supported *int* type identifiers are documented in Mapping of java.sql.Types to SQL types and in the Data types section. The *JDBCType* enums have the same names as their corresponding *int* identifiers in *java.sql.Types*.

JDBC escape syntax

JDBC provides a way of smoothing out some of the differences in the ways 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 }
```

The JDBC escape keywords are case-insensitive.

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 escape syntax is supported for a *java.sql.Statement* and a *java.sql.PreparedStatement* in addition to a *CallableStatement*.

Syntax

```
{ call statement }
```

Example

```
-- Call a Java procedure
{ call TOURS.BOOK_TOUR(?, ?) }
```

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' }
```

Examples

```
-- 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 expressions.

JDBC escape syntax for limit/offset clauses

The LIMIT escape clause can occur in a query at the point where an OFFSET/FETCH FIRST clause can appear.

See The result offset and fetch first clauses for more information.

Syntax

```
{ LIMIT rowCount [ OFFSET startRow ] }
```

The *rowCount* is a non-negative integer that specifies the number of rows to return. If *rowCount* is 0, all rows from *startRow* forward are returned.

The *startRow* is a non-negative number that specifies the number of rows to skip before returning results.

Equivalent to

```
OFFSET startRow FETCH NEXT rowCount ROWS ONLY
```

Examples

```
-- return the first two rows of sorted table t
SELECT * FROM t
ORDER BY a
{ LIMIT 2 }
-- return two rows of sorted table t, starting with the eleventh row
SELECT * FROM t
ORDER BY a
{ LIMIT 2 OFFSET 10 }
```

JDBC escape syntax for fn keyword

You can specify functions in JDBC escape syntax by using the `fn` keyword.

Syntax

```
{ fn functionCall }
```

where *functionCall* is the name of one of the scalar functions listed below. The functions are of the following types:

- Numeric functions
- String functions
- Date and time functions
- System function

Numeric functions

abs

Returns the absolute value of a number.

```
abs ( numericExpression )
```

The JDBC escape syntax `{fn abs(numericExpression)}` is equivalent to the built-in syntax `ABS(numericExpression)`. For more information, see ABS or ABSVAL function.

acos

Returns the arc cosine of a specified number.

```
acos ( number )
```

The JDBC escape syntax `{fn acos(number)}` is equivalent to the built-in syntax `ACOS(number)`. For more information, see ACOS function.

asin

Returns the arc sine of a specified number.

```
asin ( number )
```

The JDBC escape syntax `{fn asin(number)}` is equivalent to the built-in syntax `ASIN(number)`. For more information, see ASIN function.

atan

Returns the arc tangent of a specified number.

```
atan ( number )
```

The JDBC escape syntax `{fn atan(number)}` is equivalent to the built-in syntax `ATAN(number)`. For more information, see ATAN function.

atan2

Returns the arc tangent in radians of *y/x*.

```
atan2 ( y, x )
```

The JDBC escape syntax `{fn atan2(y, x)}` is equivalent to the built-in syntax `ATAN2(y, x)`. For more information, see ATAN2 function.

ceiling

Rounds the specified number up, and returns the smallest number that is greater than or equal to the specified number.

```
ceiling ( number )
```

The JDBC escape syntax `{fn ceiling(number)}` is equivalent to the built-in syntax `CEILING(number)`. For more information, see CEIL or CEILING function.

cos

Returns the cosine of a specified number.

```
cos ( number )
```

The JDBC escape syntax `{fn cos(number)}` is equivalent to the built-in syntax `COS(number)`. For more information, see COS function.

cot

Returns the cotangent of a specified number.

```
cot ( number )
```

The JDBC escape syntax `{fn cot(number)}` is equivalent to the built-in syntax `COT(number)`. For more information, see COT function.

degrees

Converts a specified number from radians to degrees.

```
degrees ( number )
```

The JDBC escape syntax `{fn degrees(number)}` is equivalent to the built-in syntax `DEGREES(number)`. For more information, see DEGREES function.

exp

Returns e raised to the power of the specified number.

```
exp ( number )
```

The JDBC escape syntax `{fn exp(number)}` is equivalent to the built-in syntax `EXP(number)`. For more information, see EXP function.

floor

Rounds the specified number down, and returns the largest number that is less than or equal to the specified number.

```
floor ( number )
```

The JDBC escape syntax `{fn floor(number)}` is equivalent to the built-in syntax `FLOOR(number)`. For more information, see FLOOR function.

log

Returns the natural logarithm (base e) of the specified number.

```
log ( number )
```

The JDBC escape syntax `{fn log(number)}` is equivalent to the built-in syntax `LOG(number)`. For more information, see LN or LOG function.

log10

Returns the base-10 logarithm of the specified number.

```
log10 ( number )
```

The JDBC escape syntax `{fn log10(number)}` is equivalent to the built-in syntax `LOG10(number)`. For more information, see LOG10 function.

mod

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

```
mod ( integerExpression, integerExpression )
```

The JDBC escape syntax `{fn mod(integerExpression, integerExpression)}` is equivalent to the built-in syntax `MOD(integerExpression, integerExpression)`. For more information, see MOD function.

pi

Returns a value that is closer than any other value to pi.

```
pi ( )
```

The JDBC escape syntax `{fn pi()}` is equivalent to the built-in syntax `PI()`. For more information, see PI function.

radians

Converts a specified number from degrees to radians.

```
radians ( number )
```

The JDBC escape syntax `{fn radians(number)}` is equivalent to the built-in syntax `RADIANS(number)`. For more information, see RADIANS function.

rand

Returns a random number given a seed number.

```
rand ( seed )
```

The JDBC escape syntax `{fn rand(seed)}` is equivalent to the built-in syntax `RAND(seed)`. For more information, see RAND function.

sign

Returns an integer that represents the sign of a specified number (+1 if the number is positive, -1 if it is negative, 0 if it is 0).

```
sign ( number )
```

The JDBC escape syntax `{fn sign(number)}` is equivalent to the built-in syntax `SIGN(number)`. For more information, see SIGN function.

sin

Returns the sine of a specified number.

```
sin ( number )
```

The JDBC escape syntax `{fn sin(number)}` is equivalent to the built-in syntax `SIN(number)`. For more information, see SIN function.

sqrt

Returns the square root of a floating-point number.

```
sqrt ( number )
```

The JDBC escape syntax `{fn sqrt(number)}` is equivalent to the built-in syntax `SQRT(number)`. For more information, see SQRT function.

tan

Returns the tangent of a specified number.

```
tan ( number )
```

The JDBC escape syntax `{fn tan(number)}` is equivalent to the built-in syntax `TAN(number)`. For more information, see TAN function.

String functions

concat

Returns the concatenation of character strings; that is, the character string formed by appending the second string to the first string. If either string is null, the result is NULL.

```
concat ( characterExpression, characterExpression )
```

The JDBC escape syntax `{fn concat(characterExpression, characterExpression)}` is equivalent to the built-in syntax `characterExpression || characterExpression`. For more information, see Concatenation operator.

lcase

Returns a string in which all alphabetic characters in the argument have been converted to lowercase.

```
lcase ( characterExpression )
```

The JDBC escape syntax `{fn lcase(characterExpression)}` is equivalent to the built-in syntax `LCASE(characterExpression)`. For more information, see LCASE or LOWER function.

length

Returns the number of characters in a character string expression.

```
length ( characterExpression )
```

The JDBC escape syntax `{fn length(characterExpression)}` is equivalent to the built-in syntax `LENGTH(characterExpression)`. For more information, see LENGTH function.

locate

Returns the position in the second *characterExpression* of the first occurrence of the first *characterExpression*. Searches from the beginning of the second *characterExpression*, unless the *startIndex* parameter is specified.

```
locate ( characterExpression, characterExpression [ , startIndex ] )
```

The JDBC escape syntax {fn locate(*characterExpression*, *characterExpression* [, *startIndex*])} is equivalent to the built-in syntax LOCATE(*characterExpression*, *characterExpression* [, *startPosition*]). For more information, see LOCATE function.

ltrim

Removes blanks from the beginning of a character string expression.

```
ltrim ( characterExpression )
```

The JDBC escape syntax {fn ltrim(*characterExpression*)} is equivalent to the built-in syntax LTRIM(*characterExpression*). For more information, see LTRIM function.

rtrim

Removes blanks from the end of a character string expression.

```
rtrim ( characterExpression )
```

The JDBC escape syntax {fn rtrim(*characterExpression*)} is equivalent to the built-in syntax RTRIM(*characterExpression*). For more information, see RTRIM function.

substring

Forms a character string by extracting *length* characters from the *characterExpression* beginning at *startIndex*. The index of the first character in the *characterExpression* is 1.

```
substring ( characterExpression, startIndex, length )
```

The JDBC escape syntax {fn substring(*characterExpression*, *startIndex*, *length*)} is equivalent to the built-in syntax SUBSTR(*characterExpression*, *startIndex*, *length*). For more information, see SUBSTR function.

ucase

Returns a string in which all alphabetic characters in the argument have been converted to uppercase.

```
ucase ( characterExpression )
```

The JDBC escape syntax {fn ucase(*characterExpression*)} is equivalent to the built-in syntax UCASE(*characterExpression*). For more information, see UCASE or UPPER function.

Date and time functions

curdate

Returns the current date.

```
curdate ( )
```

The JDBC escape syntax {fn curdate()} is equivalent to the built-in syntax CURRENT_DATE. For more information, see CURRENT_DATE function.

curtime

Returns the current time.

```
curtime ( )
```

The JDBC escape syntax {fn curtime()} is equivalent to the built-in syntax CURRENT_TIME. For more information, see CURRENT_TIME function.

hour

Returns the hour part of a time value.

```
hour ( expression )
```

The JDBC escape syntax `{fn hour(expression)}` is equivalent to the built-in syntax `HOUR(expression)`. For more information, see HOUR function.

minute

Returns the minute part of a time value.

```
minute ( expression )
```

The JDBC escape syntax `{fn minute(expression)}` is equivalent to the built-in syntax `MINUTE(expression)`. For more information, see MINUTE function.

month

Returns the month part of a date value.

```
month ( expression )
```

The JDBC escape syntax `{fn month(expression)}` is equivalent to the built-in syntax `MONTH(expression)`. For more information, see MONTH function.

second

Returns the seconds part of a time value.

```
second ( expression )
```

The JDBC escape syntax `{fn second(expression)}` is equivalent to the built-in syntax `SECOND(expression)`. For more information, see SECOND function.

TIMESTAMPADD

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.

`TIMESTAMPADD` is a JDBC escaped function and is accessible only by using the JDBC escape function syntax.

```
TIMESTAMPADD ( interval, integerExpression, timestampExpression )
```

To perform `TIMESTAMPADD` on dates and times, it is necessary to convert the dates and times 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.

Do 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

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.

The `TIMESTAMPDIFF` is a JDBC escaped function and is accessible only by using the JDBC escape function syntax.

```
TIMESTAMPDIFF ( interval, timestampExpression1, timestampExpression2 )
```

To perform `TIMESTAMPDIFF` on dates and times, it is necessary to convert the dates and times 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.

Do not put a datetime column inside a timestamp arithmetic function in WHERE clauses, because the optimizer will not use any index on the column.

year

Returns the year part of a date value.

```
year ( expression )
```

The JDBC escape syntax `{fn year(expression)}` is equivalent to the built-in syntax `YEAR(expression)`. For more information, see YEAR function.

Valid intervals for TIMESTAMPADD and TIMESTAMPDIFF

The `TIMESTAMPADD` and `TIMESTAMPDIFF` functions are 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 for the TIMESTAMPADD and TIMESTAMPDIFF escape functions

To return a timestamp value one month later than the current timestamp, use the following syntax:

```
{fn TIMESTAMPADD( SQL_TSI_MONTH, 1, CURRENT_TIMESTAMP)}
```

To return the number of weeks between now and the specified time on January 1, 2008, use the following syntax:

```
{fn TIMESTAMPDIFF(SQL_TSI_WEEK, CURRENT_TIMESTAMP,
  timestamp('2008-01-01-12.00.00.000000'))}
```

System function

user

Returns the authorization identifier or name of the current user. If there is no current user, it returns APP.

```
user ( )
```

The JDBC escape syntax `{fn user()}` is equivalent to the built-in syntax `USER`. For more information, see USER function.

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 operations.

Syntax

```
{ oj joinOperation [ joinOperation ]* }
```

Equivalent to

```
joinOperation [ joinOperation ]*
```

Examples

```
-- 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 join operation. 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 escape syntax for time formats

Derby interprets the JDBC escape syntax for time formats 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')
```

Example

```
VALUES {t '20:00:03'}
```

JDBC escape syntax for date formats

Derby interprets the JDBC escape syntax for date formats as equivalent to the correct SQL syntax for dates.

Syntax

```
{ d 'yyyy-mm-dd' }
```

Equivalent to

```
DATE('yyyy-mm-dd')
```

Example

```
VALUES {d '2010-10-19'}
```

JDBC escape syntax for timestamp formats

Derby interprets the JDBC escape syntax for timestamp formats 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-ddhh:mm:ss.f...' }
```

Equivalent to

```
TIMESTAMP('yyyy-mm-ddhh:mm:ss.f...')
```

The fractional portion of timestamp constants (*.f...*) can be omitted.

Example

```
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 (and, in some cases, the subsubprotocol). For information on how you set attributes, see Attributes of the Derby database connection URL. This section provides reference information only.

The *DriverManager.getConnection* method can take both a connection URL and a *Properties* object as arguments. If you specify any attributes both on the connection URL and in a *Properties* object, the attributes on the connection URL override the attributes in the *Properties* object.

Note: Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored.

bootPassword=key attribute

The *bootPassword=key* attribute specifies a boot password (encryption key).

The attribute specifies the key to use to:

- Encrypt a new database
- Configure an existing unencrypted database for encryption
- Boot an existing encrypted database

Specify an alphanumeric string that is at least eight characters long.

For information about data encryption, see "Configuring database encryption" in the *Derby Security Guide*.

Combining with other attributes

When you create a new database, the *bootPassword=key* attribute must be combined with the *create=true* and *dataEncryption=true* attributes.

When you configure an existing unencrypted database for encryption, the *bootPassword=key* attribute must be combined with the *dataEncryption=true* attribute. For an existing, unencrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

The *bootPassword=key* attribute can optionally be combined with the *encryptionProvider=providerName*, *encryptionAlgorithm=algorithm*, and/or *encryptionKeyLength=length* attributes.

Examples

```
-- create a new, encrypted database
jdbc:derby:newDB;create=true;dataEncryption=true;
    bootPassword=cseveryPlace
-- configure an existing unencrypted database for encryption
jdbc:derby:salesdb;dataEncryption=true;bootPassword=cseveryPlace
-- boot an existing encrypted database
jdbc:derby:encryptedDB;bootPassword=cseveryPlace
-- configure an existing encrypted database for encryption,
-- specifying the encryption algorithm
jdbc:derby:encryptedDB;dataEncryption=true;bootPassword=cseveryPlace;
```

```
    encryptionAlgorithm=DESede/CBC/NoPadding
```

collation=collation attribute

The *collation=collation* attribute is an optional attribute that specifies whether collation is based on the locale specified for the database or on Unicode codepoint collation.

The valid values for the *collation=collation* attribute are:

UCS_BASIC
: Unicode codepoint collation. This value is the default.

TERRITORY_BASED
: Based on the language specified with the *territory=ll_CC* attribute. The default collation strength for the locale is used. The default for Derby is commonly TERTIARY, in which character case is significant in searches and comparisons.

TERRITORY_BASED:PRIMARY
: Locale based with collation strength PRIMARY. Specify this value to make Derby behave similarly to many other databases, for which PRIMARY is commonly the default. PRIMARY typically means that only differences in base letters are considered significant, whereas differences in accents or case are not considered significant.

TERRITORY_BASED:SECONDARY
: Locale based with collation strength SECONDARY. SECONDARY typically means that differences in base letters or accents are considered significant, whereas differences in case are not considered significant.

TERRITORY_BASED:TERTIARY
: Locale based with collation strength TERTIARY. TERTIARY typically means that differences in base letters, accents, or case are all considered significant.

TERRITORY_BASED:IDENTICAL
: Locale based with collation strength IDENTICAL. IDENTICAL means that all differences are considered significant.

Restriction: The *collation=collation* attribute can be specified only when you create a database. You cannot specify this attribute on an existing database or when you upgrade a database.

If you specify the *collation=collation* attribute with the value TERRITORY_BASED, or one of its variants with a specific collation strength, the collation is based on the language and country codes that you specify with the *territory=ll_CC* attribute.

If you do not specify the *territory=ll_CC* attribute when you create the database, Derby uses the *java.util.Locale.getDefault* method to determine the current value of the default locale for this instance of the Java Virtual Machine (JVM).

Note: The *collation=collation* attribute applies only to user-defined tables. The system tables use the Unicode codepoint collation.

For information on how Derby handles collation, see "Creating a database with locale-based collation" and "Character-based collation in Derby" in the *Derby Developer's Guide*.

Example

The following example shows the URL to create the *MexicanDB* database. The *territory=ll_CC* attribute specifies Spanish for the language code and Mexico for the country code. The *collation=collation* attribute specifies that the collation for the database is locale based.

```
jdbc:derby:MexicanDB;create=true;territory=es_MX;collation=TERRITORY_BASED
```

create=true attribute

The *create=true* attribute 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, the attribute 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.

Database owner

When the database is created, the current authorization identifier becomes the database owner (see the *user=userName* attribute). If authentication and SQL authorization are both enabled (see "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide*), only the database owner can shut down or drop the database, encrypt it, reencrypt it with a new boot password or new encryption key, or perform a full upgrade. If authentication is not enabled, and no user is supplied, the database owner defaults to "APP", which is also the name of the default schema (see SET SCHEMA statement).

Combining with other attributes

You must specify a *databaseName* (after the subprotocol or subsubprotocol in the database connection URL) or a *databaseName=nameofDatabase* attribute with this attribute.

You can combine this attribute with other attributes. To specify a locale when creating a database, use the *territory=ll_CC* attribute.

Examples

Creating a file system database:

```
-- create a file system database
jdbc:derby:sampleDB;create=true
-- create a file system database using the databaseName attribute
jdbc:derby:;databaseName=newDB;create=true
-- create an in-memory database using the embedded driver
jdbc:derby:memory:myInMemDB;create=true
-- create an in-memory database using the databaseName attribute
jdbc:derby:;databaseName=memory:myInMemDB;create=true
-- create an in-memory database using the Network Server
jdbc:derby://localhost:1527/memory:myInMemDB;create=true
```

See "Using in-memory databases" in the *Derby Developer's Guide* for information on creating in-memory databases.

createFrom=path attribute

The *createFrom=path* attribute creates 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=logDirectoryPath* 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.

For more information about using this attribute, see "Creating a database from a backup copy" in the *Derby Server and Administration Guide*.

Combining with other attributes

Do not combine this attribute with *rollForwardRecoveryFrom*, *restoreFrom*, or *create*.

Example

```
-- create the wombat database from a backup copy
jdbc:derby:wombat;createFrom=d:/backup/wombat
```

databaseName=nameOfDatabase attribute

The *databaseName=nameOfDatabase* attribute specifies a database path name for a connection.

You can use this attribute instead of specifying the database name after the subprotocol.

The *nameOfDatabase* value can be either an absolute path name or a path name relative to *derby.system.home*. For example, `thisDB`, `databases/thisDB`, and `c:/databases/2014/january/thisDB` can all be valid values.

The path separator in the connection URL is a forward slash (`/`), even in Windows path names. The *nameOfDatabase* value cannot contain a colon (`:`), except for the colon after the drive name in a Windows path name.

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 you use a subsubprotocol to specify the database (for example, *memory* for an in-memory database), include the subsubprotocol as part of the *databaseName* attribute specification. For example:

```
jdbc:derby:;databaseName=memory:myDB
```

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 method.*

Combining with other attributes

You can combine this attribute with all other attributes.

Example

```
jdbc:derby:;databaseName=newDB;create=true
```

dataEncryption=true attribute

The *dataEncryption=true* attribute specifies data encryption on disk for a new database or to configure an existing unencrypted database for encryption.

For information about data encryption, see "Configuring database encryption" in the *Derby Security Guide*.

After you encrypt a database, you can return it to the unencrypted state by specifying the *decryptDatabase=true* attribute.

Combining with other attributes

The *dataEncryption=true* attribute must be combined with either the *bootPassword=key* attribute or the *encryptionKey=key* attribute.

With either *bootPassword=key* or *encryptionKey=key*, you have the option of also specifying the *encryptionProvider=providerName*, *encryptionAlgorithm=algorithm*, and/or *encryptionKeyLength=length* attributes.

An encryption key and a boot password use different storage mechanisms, so if, for example, you create a database using a boot password, you must continue to specify a boot password when you boot the database; you cannot switch to an encryption key. You can change either the boot password or the encryption key by specifying *newBootPassword=newPassword* or *newEncryptionKey=key*.

For an existing, unencrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

After you encrypt an existing, unencrypted database, be sure to check for *SQLWarning*s. The encryption succeeded only if there were no *SQLWarning*s or *SQLException*s.

Examples

```
-- encrypt a new database
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    bootPassword=cLo4u922sc23aPe
-- configure an existing unencrypted database for encryption
jdbc:derby:salesdb;dataEncryption=true;bootPassword=cLo4u922sc23aPe
```

decryptDatabase=true attribute

The *decryptDatabase=true* attribute returns an encrypted database to an unencrypted state.

For information about data encryption, see "Configuring database encryption" in the *Derby Security Guide*.

You must shut down the database before you decrypt it. An attempt to decrypt a booted database has no effect.

Specifying this attribute for an unencrypted database has no effect.

Combining with other attributes

The *decryptDatabase=true* attribute must be combined with either the *bootPassword=key* attribute or the *encryptionKey=key* attribute.

For an existing, encrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform decryption. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

After you decrypt the database, be sure to check for *SQLWarning*s. The decryption succeeded only if there were no *SQLWarning*s or *SQLException*s.

Examples

```
-- decrypt a database
jdbc:derby:encryptedDB;decryptDatabase=true;bootPassword=cLo4u922sc23aPe
-- decrypt a database with authentication and SQL authorization enabled
jdbc:derby:salesdb;decryptDatabase=true;user=user1;password=mypass;
    bootPassword=cLo4u922sc23aPe
```

deregister=false attribute

The *deregister=false* attribute, if set to *true* (the default), deregisters the embedded JDBC driver from the *DriverManager* after a shutdown, so that the Derby classes can be garbage-collected.

If you are running with a security manager on JDK 8 or higher, you must grant the following permission to *derby.jar* to allow the JDBC driver to be deregistered:

```
permission java.sql.SQLPermission "deregisterDriver";
```

See "Configuring Java security" in the *Derby Security Guide* for details. If you do not grant this permission when using a security manager, an error message and stack trace will appear in *derby.log* on shutdown, and the embedded JDBC driver will remain registered.

If you are running with a security manager on JDK 6 or 7, you do not need to set this permission.

You initially register the embedded driver by calling a *DriverManager* method such as *DriverManager.getDrivers()* or *DriverManager.getConnection()*.

Once the embedded driver is registered, you can shut down the Derby engine by using the *shutdown=true* connection URL attribute. If you also specify *deregister=false* with the shutdown URL, the following will happen:

- The embedded driver will remain registered.
- The Derby classes will **not** be garbage-collected.
- You can get a Derby connection by issuing a call to *DriverManager.getConnection()*.

In contrast, if you use the default setting of *deregister=true* when you shut down the database, the following will happen:

- The embedded driver will be deregistered.
- The Derby classes will be garbage-collected.
- You will have to call *Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance()* before you obtain a new Derby connection by calling *DriverManager.getConnection()*.

This attribute has no meaning if it is used with the network driver.

Combining with other attributes

This attribute is valid only when issued in conjunction with the *shutdown=true* attribute.

Examples

```
-- shut down salesDB and deregister the driver
jdbc:derby:salesDB;shutdown=true
-- shut down salesDB, but do not deregister the driver
jdbc:derby:salesDB;shutdown=true;deregister=false
```

drop=true attribute

The *drop=true* attribute removes the in-memory database specified within the database connection URL.

The attribute generates the *SQLException* 08006 if successful. If the database does not exist, it generates an error reporting that the database could not be found.

For a database for which authentication and SQL authorization are both enabled, only the database owner can drop that database.

It is not necessary to shut down the database before dropping it.

If you specify this attribute with a database that is not an in-memory database, Derby generates the *SQLException* XBM0I.

Combining with other attributes

If authentication is turned on, you must specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. If both authentication and SQL authorization are turned on, the user must be the database owner.

Examples

```
-- drop an in-memory database using the embedded driver
jdbc:derby:memory:myInMemDB;drop=true
-- drop an in-memory database using the Network Server
jdbc:derby://localhost:1527/memory:myInMemDB;drop=true
```

encryptionKey=key attribute

The *encryptionKey=key* attribute specifies an external encryption key.

The attribute specifies the external key to use to:

- Encrypt a new database
- Configure an existing unencrypted database for encryption
- Boot an existing encrypted database

Your application must provide the encryption key. The encryption key value must be a hexadecimal string at least 16 digits in length (8 bytes), and it must contain an even number of digits.

For information about data encryption, see "Configuring database encryption" in the *Derby Security Guide*.

Combining with other attributes

When creating a new database, you must combine the *encryptionKey=key* attribute with the *create=true* and *dataEncryption=true* attributes.

When you configure an existing unencrypted database for encryption, the *encryptionKey=key* attribute must be combined with the *dataEncryption=true* attribute. For an existing, unencrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

The *encryptionKey=key* attribute can optionally be combined with the *encryptionProvider=providerName*, *encryptionAlgorithm=algorithm*, and/or *encryptionKeyLength=length* attributes.

Examples

```
-- create a new encrypted database
jdbc:derby:newDB;create=true;dataEncryption=true;
   encryptionKey=6162636465666768
-- configure an existing unencrypted database for encryption
jdbc:derby:salesdb;dataEncryption=true;encryptionKey=6162636465666768
-- boot an encrypted database
jdbc:derby:encryptedDB;encryptionKey=6162636465666768
```

encryptionKeyLength=length attribute

The *encryptionKeyLength=length* attribute specifies the number of bits in the encryption key to be generated when a database is encrypted with the *bootPassword=key* attribute.

See *bootPassword=key* attribute for details.

The default encryption key length is 128.

You need to specify *encryptionKeyLength=length* only if all of the following circumstances apply:

- You are specifying a non-default encryption algorithm (the default is DES).
- The encryption algorithm you are specifying allows for more than one key length.
- You want to use a non-default key length.

For information about data encryption, see "Configuring database encryption" in the *Derby Security Guide*.

Combining with other attributes

The *encryptionKeyLength=length* attribute can be combined with either the *bootPassword=key* attribute or the *encryptionKey=key* attribute (although it is redundant if you use *encryptionKey=key*).

The *encryptionKeyLength=length* attribute may also be combined with the *encryptionProvider=providerName* and/or *encryptionAlgorithm=algorithm* attributes.

If you use *encryptionKeyLength=length* with *encryptionKey=key*, the key you specify must have the length you specify.

Examples

```
-- create a new, encrypted database
jdbc:derby:newDB;create=true;dataEncryption=true;
    encryptionKeyLength=192;encryptionAlgorithm=AES/CBC/NoPadding;
    bootPassword=Thursday
-- configure an existing unencrypted database for encryption
jdbc:derby:myDB;dataEncryption=true;
    encryptionKeyLength=168;encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=Wednesday
```

Note: If the specified algorithm does not support the specified encryption key length, Derby returns an exception.

encryptionProvider=providerName attribute

The *encryptionProvider=providerName* attribute specifies the provider for data encryption.

For information about data encryption, see "Configuring database encryption" in the *Derby Security 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

The *encryptionProvider=providerName* attribute must be combined with the *dataEncryption=true* attribute and with either the *bootPassword=key* or the *encryptionKey=key* attribute. You can also use the *encryptionAlgorithm=algorithm* attribute to specify the encryption algorithm.

For an existing, unencrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption or reencryption. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

Examples

```
-- create a new, encrypted database
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
-- configure an existing database for encryption
jdbc:derby:salesdb;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
```

encryptionAlgorithm=algorithm attribute

The *encryptionAlgorithm=algorithm* attribute specifies the algorithm for data encryption.

Use the Java conventions when you specify the algorithm. For example:

```
algorithmName/feedbackMode/padding
```

The only padding type that is allowed with Derby is *NoPadding*.

If no encryption algorithm is specified, the default value is *DES/CBC/NoPadding*.

For information about data encryption, see "Configuring database encryption" in the *Derby Security Guide*.

Combining with other attributes

The *encryptionAlgorithm=algorithm* attribute must be combined with the *dataEncryption=true* attribute and with either the *bootPassword=key* attribute or the *encryptionKey=key* attribute. You can also use the *encryptionProvider=providerName* attribute to specify the encryption provider of the algorithm.

For an existing database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption or reencryption. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

Examples

```
-- encrypt a new database
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
-- configure an existing database for encryption
jdbc:derby:salesdb;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 returns an exception.

failover=true attribute

The *failover=true* attribute stops database replication on the slave system and converts the slave database into a normal database.

If you specify the *failover=true* attribute on the master, the attribute sends the remaining log records to the slave instance and then sends a failover message to the slave. The replication functionality and the database are then shut down on the master system. If failover is successful, an exception with the error code XRE20 is thrown. Hence, when issued on the master, the *failover=true* attribute does not return a valid connection.

You may specify this attribute on the slave only if the network connection between the master and the slave is down.

When you specify this attribute on the slave, or when a failover message is sent as part of the execution of the *failover=true* attribute on the master, all transaction log chunks that have been received from the master are written to disk. The slave replication functionality is shut down, and the boot process of the database is allowed to complete. The database is now in a transaction consistent state, reflecting all transactions whose commit log records were received from the master. When issued on the slave, the *failover=true* command returns a valid connection.

The Derby instance where this command is issued must be serving the named database in replication mode.

For more information, see the topics under "Replicating databases" in the *Derby Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

If authentication is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. Authorization for the master database cannot be checked when the network connection is down, so the requirement that the user must be the database owner is not enforced.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- start failover from master using database name in subprotocol,
--   authorization
jdbc:derby:myDB;failover=true;user=mary;password=little88lamb
```

```
-- start failover using databaseName attribute, no security
jdbc:derby:;databaseName=myDB;failover=true;
```

logDevice=logDirectoryPath attribute

The *logDevice=logDirectoryPath* attribute specifies the path to the directory on which to store the database log during database creation or restoration.

Even if specified as a relative path, the *logDirectoryPath* is stored internally as an absolute path.

For more information about using this attribute, see "Using the logDevice=logDirectoryPath attribute" in the *Derby Server and Administration Guide*.

Combining with other attributes

Use in conjunction with *create*, *createFrom*, *restoreFrom*, or *rollForwardRecoveryFrom*.

Example

```
jdbc:derby:newDB;create=true;logDevice=d:/newDBlog
```

newBootPassword=newPassword attribute

The *newBootPassword=newPassword* attribute specifies a new boot password for an encrypted database.

A new encryption key is generated internally by the engine, and the key is protected using the new boot password. The newly generated encryption key encrypts the database, including the existing data. For more information about this attribute, see "Encrypting databases with a new boot password" in the *Derby Security Guide*.

Combining with other attributes

The *newBootPassword=newPassword* attribute must be combined with the *bootPassword=key* attribute.

You cannot change the encryption provider or the encryption algorithm when you use the *newBootPassword=newPassword* attribute.

For an existing encrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform reencryption. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

After you change the boot password, be sure to check for *SQLWarning*s. The change succeeded only if there were no *SQLWarning*s or *SQLException*s.

Example

```
-- specify a new boot password for a database
jdbc:derby:salesdb;bootPassword=abc1234xyz;newBootPassword=new1234xyz
```

newEncryptionKey=key attribute

The *newEncryptionKey=key* attribute specifies a new external encryption key for an encrypted database.

All of the existing data in the database is encrypted using the new encryption key, and any new data written to the database will use this key for encryption. For more information about this attribute, see "Encrypting databases with a new external encryption key" in the *Derby Security Guide*.

The encryption key value must be a hexadecimal string at least 16 digits in length (8 bytes), and it must contain an even number of digits.

Combining with other attributes

The *newEncryptionKey=key* attribute must be combined with the *encryptionKey=key* attribute.

You cannot change the encryption provider or the encryption algorithm when you use the *newEncryptionKey=key* attribute.

For an existing encrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform reencryption. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

After you change the encryption key, be sure to check for *SQLWarning*s. The change succeeded only if there were no *SQLWarning*s or *SQLException*s.

Example

```
-- specify a new encryption key for a database
jdbc:derby:salesdb;encryptionKey=6162636465666768;
    newEncryptionKey=6862636465666768
```

password=userPassword attribute

The *password=userPassword* attribute specifies a valid password for the given user name.

Combining with other attributes

Use this attribute in conjunction with the *user=userName* attribute.

Example

```
-- connect the user jack to toursDB
jdbc:derby:toursDB;user=jack;password=upTheHill
```

restoreFrom=path attribute

The *restoreFrom=path* attribute restores 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=logDirectoryPath* attribute can be used in conjunction with *restoreFrom=path* to store logs in a different location.

For more information about using this attribute, see "Restoring a database from a backup copy" in the *Derby Server and Administration Guide*.

Combining with other attributes

Do not combine this attribute with *createFrom*, *rollForwardRecoveryFrom*, or *create*.

Example

```
-- restore the wombat database from a backup copy
jdbc:derby:wombat;restoreFrom=d:/backup/wombat
```

retrieveMessageText=false attribute

The *retrieveMessageText=false* attribute, if set to *true* (the default), retrieves the message text for each *SQLException* from the server.

A stored procedure call retrieves the message text and may start a new unit of work.

Set this property to *false* if you do not want the performance impact of the procedure call or do not want to start a new unit of work.

This attribute is meaningful only if used with the network driver.

Combining with other attributes

You can combine this attribute with other attributes.

Example

```
-- do not retrieve message text for SQLExceptions
jdbc:derby://localhost:1527/salesDB;retrieveMessageText=false
```

rollForwardRecoveryFrom=path attribute

The *rollForwardRecoveryFrom=path* attribute restores a database using a backup copy and performs roll-forward recovery using archived and active logs.

To restore a database using roll-forward 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.

For more information about using this attribute, see "Roll-forward recovery" in the *Derby Server and Administration Guide*.

Combining with other attributes

Do not combine this attribute with *createFrom*, *restoreFrom*, or *create*.

Example

```
-- restore and recover the wombat database
jdbc:derby:wombat;rollForwardRecoveryFrom=d:/backup/wombat
```

securityMechanism=value attribute

The *securityMechanism=value* attribute specifies a security mechanism for client access to the Network Server.

The *value* is numeric.

Valid numeric values are:

- **9**, which specifies Encrypted UserID and Encrypted Password security. If you specify this mechanism, both the user ID and the password are encrypted. See the subsection of "Configuring Network Server authentication without SSL/TLS" entitled "Enabling the encrypted user ID and password security mechanism" in the *Derby Security Guide* for additional requirements for the use of this security mechanism.
- **3**, which specifies Clear Text Password security. Clear Text Password security is the default if you do not specify the *securityMechanism* attribute and you specify both the *user=userName* and *password=userPassword* attributes.
- **4**, which specifies User Only security. User Only security is the default if you do not specify the *securityMechanism* attribute and you specify the *user=userName* attribute but not the *password=userPassword* attribute.

Combining with other attributes

The *securityMechanism* attribute must be combined with the *user=userName* attribute.

Example

```
-- specify Encrypted UserID and Encrypted Password security
jdbc:derby://localhost/
mydb;user=myuser;password=mypassword;securityMechanism=9
```

shutdown=true attribute

The *shutdown=true* attribute shuts down the specified database if you specify a *databaseName*. (Reconnecting to the database reboots the database.)

For a database for which authentication and SQL authorization are both enabled, only the database owner can perform shutdown of that database. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

Shuts down the entire Derby system if and only if you do not specify a *databaseName*.

When you are shutting down a single database, the attribute lets Derby perform a final checkpoint on the database.

When you are shutting down a system, the attribute lets Derby perform a final checkpoint on all system databases, deregister the JDBC driver (if permitted; see *deregister=false*), 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 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.

The attribute is 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.

Combining with other attributes

This attribute may be combined with the *deregister=false* attribute.

When you shut down a database, if authentication is turned on, you must specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. If both authentication and SQL authorization are turned on, the user must be the database owner.

When you shut down the Derby system, if both authentication and SQL authorization are turned on, you must specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes.

Examples

```
-- shut down entire system
jdbc:derby:;shutdown=true
-- shut down salesDB (authentication not enabled)
jdbc:derby:salesDB;shutdown=true
-- shut down an in-memory database using the embedded driver
jdbc:derby:memory:myInMemDB;shutdown=true
-- shut down an in-memory database using the Network Server
jdbc:derby://localhost:1527/memory:myInMemDB;shutdown=true
```

slaveHost=hostname attribute

The *slaveHost=hostname* attribute specifies the system that will serve as the slave for database replication.

For more information, see the topics under "Replicating databases" in the *Derby Server and Administration Guide*.

Combining with other attributes

This attribute must be specified in conjunction with the *startMaster=true* attribute. It may be specified in conjunction with the *startSlave=true* attribute; if it is not, the default value is `localhost`.

This attribute may be specified only in conjunction with the other attributes permitted with the *startMaster=true* and *startSlave=true* attributes.

Examples

For examples, see *startMaster=true* and *startSlave=true*.

slavePort=portValue attribute

The *slavePort=portValue* attribute specifies the port that the slave system will use in database replication.

For more information, see the topics under "Replicating databases" in the *Derby Server and Administration Guide*.

Combining with other attributes

This attribute may be specified in conjunction with the *startMaster=true* attribute and the *startSlave=true* attribute. If it is not specified, the default port value is `4851`.

This attribute may be specified only in conjunction with the other attributes permitted with the *startMaster=true* and *startSlave=true* attributes.

Examples

For examples, see *startMaster=true* and *startSlave=true*.

ssl=sslMode attribute

The *ssl=sslMode* attribute specifies the SSL mode of the client.

The *sslMode* can be `basic`, `peerAuthentication`, or `off` (the default). See "Configuring SSL/TLS" in the *Derby Security Guide* for details.

Combining with other attributes

May be combined with all other attributes.

Example

```
-- connect to mydb with basic SSL encryption
jdbc:derby://localhost/mydb;ssl=basic
```

startMaster=true attribute

The *startMaster=true* attribute starts replication of a database in master mode.

Before you specify this attribute, you must cleanly shut down the database on the master system, perform a file system copy of the database to the slave system, and specify the *startSlave=true* attribute. For details, see the topic "Starting and running replication" under "Replicating databases" in the *Derby Server and Administration Guide*.

If the master database is already booted and any unlogged operations are running when the user specifies *startMaster=true*, the attempt to start the master fails and an error message appears.

For more information on replication, see the other topics under "Replicating databases" in the *Derby Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

You must specify this attribute in conjunction with the *slaveHost=hostname* attribute. You may also specify this attribute in conjunction with the *slavePort=portValue* attribute. If you do not specify the *slavePort=portValue* attribute, the default port value is 4851.

If authentication is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. If both authentication and SQL authorization are turned on, the user must be the database owner.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- start master using database name in subprotocol, default slave
--  port, authorization
jdbc:derby:myDB;startMaster=true;slaveHost=elsewhere;user=mary;
   password=little88lamb
```

```
-- start master using databaseName attribute, non-default slave
--  port, no security
jdbc:derby:;databaseName=myDB;startMaster=true;slaveHost=elsewhere;
   slavePort=4852
```

startSlave=true attribute

The *startSlave=true* attribute starts replication of a database in slave mode.

Before you specify this attribute, you must cleanly shut down the database on the master system and then perform a file system copy of the database to the slave system.

The *startSlave=true* attribute does the following:

1. Partially boots the specified database
2. Starts to listen on the specified port and accepts a connection from the master
3. Hangs until the master has connected to it
4. Reports the startup status to the caller (whether it has started, and if not, why not)
5. Continually receives chunks of the transaction log from the master and applies the operations in the transaction log to the slave database

If replication is started successfully, an exception with the error code XRE08 is thrown. Hence, the *startSlave=true* attribute does not return a valid connection.

For more information, see the topics under "Replicating databases" in the *Derby Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

You may specify this attribute in conjunction with the *slaveHost=hostname* attribute. If you do not specify the *slaveHost=hostname* attribute, the default value is `localhost`.

You may also specify this attribute in conjunction with the *slavePort=portValue* attribute. If you do not specify the *slavePort=portValue* attribute, the default port value is 4851.

If authentication is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. If both authentication and SQL authorization are turned on, the user must be the database owner.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- start slave using database name in subprotocol, default slave host
--    and port, authorization
jdbc:derby:myDB;startslave=true;user=mary;password=little88lamb
```

```
-- start slave using databaseName attribute, non-default slave host
--   and port, no security
jdbc:derby:;databaseName=myDB;startSlave=true;slaveHost=localhost;
   slavePort=4852
```

stopMaster=true attribute

The *stopMaster=true* attribute stops database replication on the master system.

This attribute sends a stop-slave message to the slave system if the network connection is working. Then it shuts down all replication-related functionality, without shutting down the specified database.

The Derby instance where this attribute is specified must be the replication master for the specified database.

For more information, see the topics under "Replicating databases" in the *Derby Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

If authentication is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. If both authentication and SQL authorization are turned on, the user must be the database owner.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- stop master using database name in subprotocol, authorization
jdbc:derby:myDB;stopMaster=true;user=mary;password=little88lamb
```

```
-- stop master using databaseName attribute, no security
jdbc:derby:;databaseName=myDB;stopMaster=true;
```

stopSlave=true attribute

The *stopSlave=true* attribute stops database replication on the slave system.

You can specify this connection URL attribute only if the network connection between the master and slave systems is down. If the network connection is working, the slave system accepts commands only from the master, so you must specify the *stopMaster=true* attribute on the master system to stop replication on both the master and slave systems.

When this attribute is specified, or when a stop-slave message is sent as part of the execution of the *stopMaster=true* attribute, all transaction log chunks that have been received from the master are written to disk. Both the slave replication functionality and the database are then shut down.

The Derby instance where this attribute is specified must be serving the specified database in replication slave mode.

For more information, see the topics under "Replicating databases" in the *Derby Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

If authentication is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. Authorization for the master database cannot be checked when the network connection is down, so the requirement that the user must be the database owner is not enforced.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- stop slave from master using database name in subprotocol,
--   authorization
jdbc:derby:myDB;stopSlave=true;user=mary;password=little88lamb
```

```
-- stop slave using databaseName attribute, no security
jdbc:derby:;databaseName=myDB;stopSlave=true;
```

territory=ll_CC attribute

The *territory=ll_CC* attribute associates a non-default locale with a database at database creation time.

Setting the *territory=ll_CC* attribute overrides the default system locale for that database. To find the default system locale, use the *java.util.Locale.getDefault* method.

Specify a locale 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 lowercase letters that conform to ISO 639-1. The following table shows some examples.

Table 147. 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.loc.gov/standards/iso639-2/php/code_list.php*.

Country codes consist of two uppercase letters that conform to ISO 3166. The following table shows some examples.

Table 148. Sample country codes

Country Code	Description
DE	Germany
US	United States
ES	Spain

Country Code	Description
MX	Mexico
JP	Japan

A copy of ISO 3166 can be found at *http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html*.

Combining with other attributes

Use the *territory=ll_CC* attribute only when you create a database.

Example

In the following example, the new database has a locale of Spanish language and Mexican nationality.

```
jdbc:derby:MexicanDB;create=true;territory=es_MX
```

You can use the *collation=collation* attribute with the *territory=ll_CC* attribute to specify that collation is based on the locale, instead of being based on Unicode codepoint collation. For more information, see "Creating a database with locale-based collation", "Creating a case-insensitive database", and "Character-based collation in Derby" in the *Derby Developer's Guide*.

traceDirectory=path attribute

The *traceDirectory=path* attribute specifies a directory to which the Derby Network Client will send JDBC trace information.

If the program or session has multiple connections, the Network Client creates a separate file for each connection. By default, the files are named `_driver_0`, `_driver_1`, and so on. Use the *traceFile=path* attribute to specify a file name for the trace file.

If the directory does not exist, Derby issues an error message. If you do not specify an absolute path name, the directory is assumed to be relative to the current directory.

For more information about tracing, see "Network client tracing" in the *Derby Server and Administration Guide*. See *traceFile=path*, *traceFileAppend=true*, and *traceLevel=value* for other attributes related to tracing.

Combining with other attributes

You can combine this attribute with other attributes.

Examples

```
-- enable tracing on an existing database that will have multiple
--   connections
jdbc:derby://localhost:1527/mydb;traceDirectory=/home/mydir/mydbtracedir
-- specify a trace file name within the directory
jdbc:derby://localhost:1527/mydb;
   traceDirectory=/home/mydir/mydbtracedir;traceFile=trace.out
-- append to the default trace file
jdbc:derby://localhost:1527/mydb;
   traceDirectory=/home/mydir/mydbtracedir;traceFileAppend=true
```

traceFile=path attribute

The *traceFile=path* attribute specifies a file to which the Derby Network Client will send JDBC trace information.

If you do not specify an absolute path name, the file is placed in the *derby.system.home* directory (see "Defining the system directory" in the *Derby Developer's Guide* for details).

If you specify both *traceFile=path* and *traceFileAppend=true*, the output is appended to the specified file, if it exists. If you specify *traceFile=path* but do not specify *traceFileAppend=true*, any previous version of the file of the file is overwritten.

For more information about tracing, see "Network client tracing" in the *Derby Server and Administration Guide*. See *traceDirectory=path* and *traceLevel=value* for other attributes related to tracing.

Combining with other attributes

You can combine this attribute with other attributes.

Example

```
-- enable tracing on a new database
jdbc:derby://localhost:1527/mydb;create=true;traceFile=trace.out
```

traceFileAppend=true attribute

The *traceFileAppend=true* attribute specifies that the Derby Network Client should append JDBC trace information to a trace file.

The file can be specified by the *traceFile=path* attribute. If you do not specify a trace file but you specify the *traceDirectory=path* attribute, the trace information is appended to the default file. If you do not specify *traceFileAppend=true*, any previous version of the trace file is overwritten.

For more information about tracing, see "Network client tracing" in the *Derby Server and Administration Guide*. See *traceDirectory=path* and *traceLevel=value* for other attributes related to tracing.

Combining with other attributes

This attribute must be specified in conjunction with either the *traceFile=path* attribute or the *traceDirectory=path* attribute. You can also combine this attribute with other attributes.

Examples

```
-- enable tracing on an existing database, appending to the
--   specified file
jdbc:derby://localhost:1527/mydb;traceFile=trace.out;
    traceFileAppend=true
-- enable tracing on an existing database, appending to the default file
--   within the specified directory, relative to the Derby home directory
jdbc:derby://localhost:1527/mydb;traceDirectory=mytracedir;
    traceFileAppend=true
```

traceLevel=value attribute

If tracing is enabled, the *traceLevel=value* attribute specifies the level of tracing to be used by the Derby Network Client.

The *value* is numeric. If you do not specify a trace level, the default is TRACE_ALL.

For more information about tracing, see "Network client tracing" in the *Derby Server and Administration Guide*. See *traceFile=path*, *traceFileAppend=true*, and *traceDirectory=path* for other attributes related to tracing.

Tracing levels

The following table shows the available tracing levels and their values.

Table 149. Available tracing levels and values

Trace Level	Hex Value	Decimal Value
org.apache.derby.jdbc.ClientDataSource.TRACE_NONE	0x0	0
org.apache.derby.jdbc.ClientDataSource.TRACE_CONNECTION_CALL	0x1	1
org.apache.derby.jdbc.ClientDataSource.TRACE_STATEMENT_CALLS	0x2	2
org.apache.derby.jdbc.ClientDataSource.TRACE_RESULT_SET_CALL	0x4	4
org.apache.derby.jdbc.ClientDataSource.TRACE_DRIVER_CONFIGUR	0x10	16
org.apache.derby.jdbc.ClientDataSource.TRACE_CONNECTS	0x20	32
org.apache.derby.jdbc.ClientDataSource.TRACE_PROTOCOL_FLOWS	0x40	64
org.apache.derby.jdbc.ClientDataSource.TRACE_RESULT_SET_META	0x80	128
org.apache.derby.jdbc.ClientDataSource.TRACE_PARAMETER_META	0x100	256
org.apache.derby.jdbc.ClientDataSource.TRACE_DIAGNOSTICS	0x200	512
org.apache.derby.jdbc.ClientDataSource.TRACE_XA_CALLS	0x800	2048
org.apache.derby.jdbc.ClientDataSource.TRACE_ALL	0xFFF	-1

Derby provides two ClientDataSource implementations. You can use the `org.apache.derby.jdbc.ClientDataSource` class on all supported Java SE versions except Java SE 8 Compact Profile 2. On Java SE 8 Compact Profile 2, you must use the `org.apache.derby.jdbc.BasicClientDataSource40` class. See JDBC support for Java SE 8 Compact Profiles for more information.

To specify more than one trace level, use one of the following techniques:

- If you are using the `ij` tool, add the decimal values together and specify the sum. For example, to trace both PROTOCOL flows and connection calls, add the values for TRACE_PROTOCOL_FLOWS (64) and TRACE_CONNECTION_CALLS (1). Specify the sum, the value 65.
- If you are running a JDBC program, do one of the following:
  - Use bitwise OR operators (|) with two or more trace values. For example, to trace protocol flows and connection calls, specify this value for traceLevel:

    ```
    TRACE_PROTOCOL_FLOWS | TRACE_CONNECTION_CALLS
    ```
  - Use a bitwise complement operator (~) with a trace value to specify all except a certain trace. For example, to trace everything except protocol flows, specify this value for traceLevel:

    ```
    ~TRACE_PROTOCOL_FLOWS
    ```

Combining with other attributes

If you specify this attribute, you must also specify either the *traceFile=path* attribute or the *traceDirectory=path* attribute.

Example

```
-- enable tracing on a new database
jdbc:derby://localhost:1527/mydb;create=true;traceFile=trace.out;
    traceLevel=65
```

upgrade=true attribute

The *upgrade=true* attribute upgrades a database that was created using an earlier version of Derby to the current version of Derby, then connects to it.

If the database does not exist, an error appears in the error log and the connection attempt fails with an *SQLException* indicating that the database cannot be found.

This operation performs a full upgrade, as defined in "Upgrading a database" in the *Derby Developer's Guide*. For more information about upgrades, see the other topics under "Upgrades" in the *Derby Developer's Guide*.

For a database for which authentication and SQL authorization are both enabled, only the database owner can perform a full upgrade. See "Configuring user authentication" and "Configuring user authorization" in the *Derby Security Guide* for more information.

Note: You cannot perform a full upgrade on a database already booted in soft upgrade mode. If a database is already booted in soft upgrade mode, the *upgrade=true* attribute will have no effect. If a database is already booted in soft upgrade mode, you can first shutdown the database with the *shutdown=true* attribute and then connect with *upgrade=true* to perform the upgrade.

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 cannot combine this attribute with the *collation* or *territory=ll_CC* attributes.

Examples

```
jdbc:derby:sampleDB;upgrade=true

jdbc:derby:;databaseName=sampleDB;upgrade=true;
```

user=userName attribute

The *user=userName* attribute specifies a valid user name for the system, in conjunction with a password.

A valid user name and password are required when user authentication is turned on.

Combining with other attributes

Use this attribute in conjunction with the *password=userPassword* attribute.

Example

```
-- connect the user jill to toursDB
jdbc:derby:toursDB;user=jill;password=toFetchAPail
```

Creating a connection without specifying 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.

Example

```
jdbc:derby:mydb
```

Derby property reference

This section provides reference information on Derby properties. For information on using these properties, see "Working with Derby properties" in the *Derby Developer's Guide*.

Scope of Derby properties

A property in Derby can have system-wide scope, database-wide scope, or both.

- *system-wide*

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

  - Set programmatically

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

    The *derby.properties* file is an optional file that can be created to set properties at the system level when the Derby driver is loaded. Derby looks for this file in the directory defined by the *derby.system.home* property. Any property except *derby.system.home* can be set by including it in the *derby.properties* file.
- *database-wide*

  A database-wide property is stored in a database and is valid for that specific database only.

Note: Database-wide properties are stored in the database and are simpler and safer for deployment. System-wide properties can be more practical during the development process.

For more information about scopes, precedence, and persistence, see "Properties overview" in the *Derby Developer's Guide*.

Dynamic and static properties

A property in Derby can be dynamic or static.

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

Some properties are static, which means changes to their values will not take effect while Derby is running. You must restart or set them before (or while) starting Derby.

Note: Properties set in the *derby.properties* file and on the command line of the application that boots Derby are *always* static, because Derby reads this file and those parameters only at startup.

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

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

Derby properties

The Derby properties are used for configuring the system and database, as well as for diagnostics such as logging statements, and monitoring and tracing locks.

The table later in this topic lists all the core Derby properties.

Derby also supports a number of properties that are specific to the Network Server. For information about these properties, see "Setting Network Server properties" in the *Derby Server and Administration Guide*.

For information on how to set Derby properties, see "Setting Derby properties" in the *Derby Developer's Guide*.

Note: When setting properties that have boolean values, be sure to trim extra spaces around the word *true*. Extra spaces around the word *true* cause the property to be set to false, regardless of its previous setting.

To disable or turn off a database-wide property setting, set its value to null. To determine the result of this action, recall that the search order for properties is as follows (as stated in "Precedence of properties" in the *Derby Developer's Guide*).

1. System-wide properties set programmatically (as a command-line option to the JVM when starting the application or within application code); not consulted if *derby.database.propertiesOnly* is set to true
2. Database-wide properties
3. System-wide properties set in the *derby.properties* file; not consulted if *derby.database.propertiesOnly* is set to true

Setting the database-wide property to null has the effect of removing the property from the list of database properties and restoring the system property setting from *derby.properties* if there is one. As always, if no value can be determined from the search, the built-in default applies.

For example, the following code fragment turns off a previous database-wide setting of the *derby.database.fullAccessUsers* property:

```
Statement s = conn.createStatement();
s.executeUpdate("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
    "'derby.database.fullAccessUsers', null)");
```

If the property is a static one, the null setting does not take effect until you reboot the database. Moreover, the static property *derby.database.sqlAuthorization* cannot be disabled after it has been enabled, even with a reboot.

The following table summarizes the general Derby properties. In the Scope column of this table, S stands for system-wide, D stands for database-wide, and C indicates the value persists with newly created conglomerates.

Table 150. Derby properties

Property	Scope	Dynamic or Static
derby.authentication.builtin.algorithm	S, D	Dynamic
derby.authentication.builtin.iterations	S, D	Dynamic
derby.authentication.builtin.saltLength	S, D	Dynamic
derby.authentication.ldap.searchAuthDN	S, D	Static
derby.authentication.ldap.searchAuthPW	S, D	Static
derby.authentication.ldap.searchBase	S, D	Static

Property	Scope	Dynamic or Static
derby.authentication.ldap.searchFilter	S, D	Static
derby.authentication.native.passwordLifetim	S, D	Static
derby.authentication.native.passwordLifetim	S, D	Static
derby.authentication.provider	S, D	Static
derby.authentication.server	S, D	Static
derby.connection.requireAuthentication	S, D	Static
derby.database.classpath	D	Dynamic
derby.database.defaultConnectionMode	S, D	See the main page for this property for information about when changes to the property are dynamic
derby.database.forceDatabaseLock	S	Static
derby.database.fullAccessUsers	S, D	See the main page for this property for information about when changes to the property are dynamic
derby.database.noAutoBoot	D	Static
derby.database.propertiesOnly	D	Dynamic
derby.database.readOnlyAccessUsers	S, D	See the main page for this property for information about when changes to the property are dynamic
derby.database.sqlAuthorization	S, D	Static
derby.infolog.append	S	Static
derby.jdbc.xaTransactionTimeout	S, D	Dynamic
derby.language.logQueryPlan	S	Static
derby.language.logStatementText	S, D	Static
derby.language.sequence.preallocator	S, D	Static
derby.language.statementCacheSize	S, D	Static
derby.locks.deadlockTimeout	S, D	Dynamic
derby.locks.deadlockTrace	S, D	Dynamic
derby.locks.escalationThreshold	S, D	Dynamic
derby.locks.monitor	S, D	Dynamic
derby.locks.waitTimeout	S, D	Dynamic
derby.replication.logBufferSize	S	Static
derby.replication.maxLogShippingInterval	S	Static
derby.replication.minLogShippingInterval	S	Static
derby.replication.verbose	S	Static
derby.storage.indexStats.auto	S, D	Static

Property	Scope	Dynamic or Static
derby.storage.indexStats.log	S, D	Static
derby.storage.indexStats.trace	S, D	Static
derby.storage.initialPages	C	Static
derby.storage.minimumRecordSize	S, D, C	Dynamic
derby.storage.pageCacheSize	S	Static
derby.storage.pageReservedSpace	S, D, C	Dynamic
derby.storage.pageSize	S, D, C	Dynamic
derby.storage.rowLocking	S, D	Static
derby.storage.tempDirectory	S, D	Static
derby.storage.useDefaultFilePermissions	S	Dynamic
derby.stream.error.extendedDiagSeverityLe	S	Static
derby.stream.error.field	S	Static
derby.stream.error.file	S	Static
derby.stream.error.logBootTrace	S	Static
derby.stream.error.logSeverityLevel	S	Static
derby.stream.error.method	S	Static
derby.stream.error.rollingFile.count	S	Static
derby.stream.error.rollingFile.limit	S	Static
derby.stream.error.rollingFile.pattern	S	Static
derby.stream.error.style	S	Static
derby.system.bootAll	S	Static
derby.system.durability	S	Static
derby.system.home	S	Static
derby.user.UserName	S, D	Dynamic
DataDictionaryVersion	D	Neither; value can be retrieved, but not set

There are additional properties associated with the Derby tools. For more information about tool-specific properties, see the *Derby Tools and Utilities Guide*.

derby.authentication.builtin.algorithm

The *derby.authentication.builtin.algorithm* property specifies the message digest algorithm to use to protect the passwords that are stored in the database when using NATIVE authentication.

The value is the name of a message digest algorithm available from one of the Java Cryptography Extension (JCE) providers registered in the JVM. Some examples of valid values are MD5, SHA-256, and SHA-512.

The specified algorithm will be applied on the concatenation of the user name and the password before it is stored in the database.

Syntax

```
derby.authentication.builtin.algorithm=algorithm
```

If the value of *algorithm* is NULL or an empty string, SHA-1 will be used on the password only.

Default

For a newly created database, the default value is SHA-256, if that algorithm is available. If SHA-256 is not available, the default is SHA-1.

Example

```
-- system-wide property
derby.authentication.builtin.algorithm=SHA-512

-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.builtin.algorithm', 'SHA-512');
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.authentication.builtin.iterations

The *derby.authentication.builtin.iterations* property specifies the number of times to apply the hash function (which is specified by the *derby.authentication.builtin.algorithm* property) on the credentials.

Iteration slows down attackers by forcing them to spend more time calculating hashes. See *derby.authentication.builtin.algorithm* for more information.

This property is in effect only if NATIVE authentication is specified by the *derby.authentication.provider* property and if the *derby.authentication.builtin.algorithm* property has a non-null value.

Syntax

```
derby.authentication.builtin.iteration=number_of_iterations
```

The minimum value is 1.

Default

1000.

Example

```
-- system-wide property
derby.authentication.builtin.iterations=2000

-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.builtin.iterations', '2000');
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.authentication.builtin.saltLength

The *derby.authentication.builtin.saltLength* property specifies the number of bytes of random salt that will be added to users' credentials before hashing them.

Random salt has the effect of making it difficult for attackers to decode passwords by constructing rainbow tables.

This property is in effect only if NATIVE authentication is specified by the *derby.authentication.provider* property and if the *derby.authentication.builtin.algorithm* property has a non-null value.

Syntax

```
derby.authentication.builtin.saltLength=number_of_bytes
```

Default

16.

Example

```
-- system-wide property
derby.authentication.builtin.saltLength=32

-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.builtin.saltLength', '32');
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.authentication.ldap.searchAuthDN

Along with *derby.authentication.ldap.searchAuthPW*, the *derby.authentication.ldap.searchAuthDN* property indicates how Derby should bind with the LDAP directory server to do searches for user DN (distinguished name).

This property specifies the DN; *derby.authentication.ldap.searchAuthPW* specifies the password to use for the search.

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

For more information about LDAP user authentication, see "Configuring LDAP authentication" in the *Derby Security Guide*.

Syntax

```
derby.authentication.ldap.searchAuthDn=DN
```

Default

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

Example

```
-- system-wide property
derby.authentication.ldap.searchAuthDn=
    cn=guest,o=example.com
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.ldap.searchAuthDn',
    'cn=guest,o=example.com')
```

Dynamic or static

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

derby.authentication.ldap.searchAuthPW

Along with *derby.authentication.ldap.searchAuthDN*, the *derby.authentication.ldap.searchAuthPW* property indicates how Derby should bind with the directory server to do searches in order to retrieve a fully qualified user DN (distinguished name).

This property specifies the password; *derby.authentication.ldap.searchAuthDN* specifies the DN to use for the search.

For more information about LDAP user authentication, see "Configuring LDAP authentication" in the *Derby Security Guide*.

Default

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

Example

```
-- system-wide property
derby.authentication.ldap.searchAuthPW=guestPassword
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.ldap.searchAuthPW',
    'guestPassword')
```

Dynamic or static

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

derby.authentication.ldap.searchBase

The *derby.authentication.ldap.searchBase* property specifies the root DN of the point in your hierarchy from which to begin a guest or anonymous search for the user's DN.

For example:

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

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

For more information about LDAP user authentication, see "Configuring LDAP authentication" in the *Derby Security Guide*.

Example

```
-- system-wide property
derby.authentication.ldap.searchBase=
    ou=people,o=example.com
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.ldap.searchBase',
    'ou=people,o=example.com')
```

Dynamic or static

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

derby.authentication.ldap.searchFilter

The *derby.authentication.ldap.searchFilter* property specifies the search filter to use to determine what constitutes a user (and other search predicate) for Derby searches for a full DN during user authentication.

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

For more information about LDAP user authentication, see "Configuring LDAP authentication" in the *Derby Security Guide*.

Syntax

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

Default

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

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

Example

```
-- system-wide properties
derby.authentication.ldap.searchFilter=objectClass=person
## people in the marketing department
## Derby automatically adds (uid=<userName>)
derby.authentication.ldap.searchFilter=(&(ou=Marketing)
    (objectClass=person))
## all people but those in marketing
## Derby automatically adds (uid=<userName>)
derby.authentication.ldap.searchFilter=(&(!(ou=Marketing)
    (objectClass=person))
## map %USERNAME% to user, not uid
derby.authentication.ldap.searchFilter=(&((ou=People)
(user=%USERNAME%))
## cache user DNs locally and use the default for others
derby.authentication.ldap.searchFilter=derby.user

-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.ldap.searchFilter',
    'objectClass=person')
```

Dynamic or static

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

derby.authentication.native.passwordLifetimeMillis

The *derby.authentication.native.passwordLifetimeMillis* property specifies the number of milliseconds a NATIVE authentication password remains valid after being created, reset, or modified.

If the value is less than or equal to zero, the password never expires.

To avoid locking out the super-user, the password of the database owner of a credentials database never expires.

If a connection attempt is made when the password's remaining lifetime is less than a proportion of the maximum lifetime, a warning is issued. The proportion is specified by the *derby.authentication.native.passwordLifetimeThreshold* property.

Syntax

```
derby.authentication.native.passwordLifetimeMillis=millis
```

Default

A number of milliseconds equal to 31 days (2,678,400,000).

Example

```
-- system-wide property
derby.authentication.native.passwordLifetimeMillis=5356800000

-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.native.passwordLifetimeMillis', '5356800000');
```

Dynamic or static

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

derby.authentication.native.passwordLifetimeThreshold

The *derby.authentication.native.passwordLifetimeThreshold* property specifies the threshold that triggers a connection-time warning that a password is about to expire.

The threshold must be a DOUBLE value greater than 0.0.

To avoid locking out the super-user, the password of the database owner of a credentials database never expires.

A warning is raised when a user logs in and the remaining lifetime of that user's password is less than this proportion of the maximum password lifetime. That is, Derby raises a warning when the remaining lifetime of a password is less than `(derby.authentication.native.passwordLifetimeThreshold * derby.authentication.native.passwordLifetimeMillis)`.

To set the lifetime of the password, use the *derby.authentication.native.passwordLifetimeMillis* property.

Syntax

```
derby.authentication.native.passwordLifetimeThreshold=proportion
```

Default

0.125 (1/8).

Example

```
-- system-wide property
derby.authentication.native.passwordLifetimeThreshold=0.2

-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
```

```
    'derby.authentication.native.passwordLifetimeThreshold', '0.2');
```

Dynamic or static

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

derby.authentication.provider

The *derby.authentication.provider* property specifies the authentication provider for Derby user authentication.

Legal values include:

- NATIVE:*credentialsDB*

  NATIVE authentication using *credentialsDB*, a dedicated database, to store user credentials. This value must be set by using system-wide Java Virtual Machine (JVM) properties or by using the *derby.properties* file; it cannot be set in the database by using the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure. When this system-wide value is set, *credentialsDB* is used to authenticate all operations. Individual databases can override this directive by specifying their own value for *derby.authentication.provider*.

  The value of *credentialsDB* must be a valid name for a database.
- NATIVE:*credentialsDB*:LOCAL

  NATIVE authentication using *credentialsDB* for system-wide operations, but using an individual database's SYSUSERS system table to authenticate connections to that database. This value must be set by using system-wide JVM properties or by using the *derby.properties* file; it cannot be set in the database by using the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY procedure.
- LDAP

  An external LDAP directory service.
- A complete Java class name

  A user-defined class that provides user authentication.

For more information about these settings, see "Configuring NATIVE authentication" and "Configuring LDAP authentication" in the *Derby Security Guide*.

To enable any Derby user authentication setting other than NATIVE, you must set the *derby.connection.requireAuthentication* property to true. If you specify NATIVE authentication, Derby behaves as if the *derby.connection.requireAuthentication* and *derby.database.sqlAuthorization* properties were also set.

When using NATIVE authentication, you can also set the following related properties:

- *derby.authentication.native.passwordLifetimeMillis*
- *derby.authentication.native.passwordLifetimeThreshold*

When using NATIVE authentication, the database owner calls the `SYSCS_UTIL.SYSCS_CREATE_USER` system procedure to create users, and can also call the following additional user management procedures:

- `SYSCS_UTIL.SYSCS_DROP_USER`
- `SYSCS_UTIL.SYSCS_RESET_PASSWORD`

When using NATIVE authentication, any user can call the `SYSCS_UTIL.SYSCS_MODIFY_PASSWORD` system procedure to change that user's password.

For more information about user authentication, see "Configuring user authentication" in the *Derby Security Guide*.

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

- *derby.authentication.server*

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

- *derby.authentication.ldap.searchAuthDN*
- *derby.authentication.ldap.searchAuthPW*
- *derby.authentication.ldap.searchFilter*
- *derby.authentication.ldap.searchBase*

Alternatively, you can write your own class to provide a different external authentication service. This class must implement the public interface *org.apache.derby.authentication.UserAuthenticator* and throw exceptions of the type *java.sql.SQLException* where appropriate. Using a user-defined class makes Derby adaptable to various naming and directory services. For example, the class could allow Derby to hook up to an existing user authentication service that uses any of the standard directory and naming service providers to JNDI.

Syntax

```
derby.authentication.provider={ NATIVE:credentialsDB |
                                NATIVE:credentialsDB:LOCAL |
                                LDAP |
                                classProviderName }
```

Default

No authentication.

Example

```
-- system-wide property
derby.authentication.provider=NATIVE:MyCredsDB:LOCAL
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.provider',
    'LDAP')
```

Dynamic or static

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

derby.authentication.server

The *derby.authentication.server* property specifies the location of the external directory service that provides user authentication for the Derby system as defined with the *derby.authentication.provider* property.

For LDAP, specify the host name and port number. See *derby.authentication.provider* for more information.

The server must be known on the network.

For more information about external user authentication, see "Configuring LDAP authentication" in the *Derby Security Guide*.

Default

Not applicable. Note that if the protocol type is unspecified, it defaults to LDAP.

Syntax

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

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

Example

```
-- system-wide property
##LDAP example
derby.authentication.server=godfrey:9090
##LDAP example
derby.authentication.server=ldap://godfrey:9090
##LDAP example
derby.authentication.server=//godfrey:9090
##LDAP over SSL example
derby.authentication.server=ldaps://godfrey:636/
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.server',
    'godfrey:9090')
```

Dynamic or static

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

derby.connection.requireAuthentication

The *derby.connection.requireAuthentication* property turns on user authentication for Derby.

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

Derby uses the type of user authentication specified with the *derby.authentication.provider* property.

If the *derby.authentication.provider* property specifies NATIVE authentication, Derby behaves as if *derby.connection.requireAuthentication* were set to *TRUE*, regardless of how *derby.connection.requireAuthentication* has been set by other means.

For more information about user authentication, see "Configuring user authentication" in the *Derby Security Guide*.

Default

False.

By default, no user authentication is required.

Example

```
-- system-wide property
derby.connection.requireAuthentication=true
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.connection.requireAuthentication',
    'true')
```

Dynamic or static

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

derby.database.classpath

The *derby.database.classpath* property specifies the classpath that Derby should use when searching for jar files in a database.

This property must be set to enable Derby to load classes from jar files installed with the SQLJ.INSTALL_JAR system procedure.

Make sure to do the following:

- Separate jar files with a colon (`:`).
- Use fully qualified identifiers for the jar files (schema name and jar name).
- Set the property as a database-level property for the database.

Derby searches the user's classpath before it searches the jar files specified by the *derby.database.classpath* property setting. To force Derby to search the database only, remove the classes from the user classpath.

Derby searches for classes and resources in the order specified by the property setting.

For more information, see the section "Loading classes from a database" in the *Derby Developer's Guide*. For reference information on system procedures for storing jar files in a database, see "System procedures for storing jar files in a database."

Syntax

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.classpath',
    'colonSeparatedJarFiles')
```

Example

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.classpath',
    'APP.ToursLogic:APP.ACCOUNTINGLOGIC')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.defaultConnectionMode

The *derby.database.defaultConnectionMode* property, one of the user authorization properties, defines the default connection mode for users of the database or system for which this property is set.

The possible values (which are case-insensitive) are:

- *noAccess*

  Disallows connections.
- *readOnlyAccess*

  Grants read-only connections.
- *fullAccess*

Grants full access.

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

Note: It is possible to configure a database so that it cannot be changed (or even accessed) using this property. If you set this property to *noAccess* or *readOnlyAccess*, be sure to allow at least one user full access. See *derby.database.fullAccessUsers* and *derby.database.readOnlyAccessUsers*.

For more information about user authorization, see "Configuring user authorization" in the *Derby Security Guide*.

Syntax

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode',
    '{ noAccess | readOnlyAccess | fullAccess}')
```

Example

```
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode', 'noAccess')
-- system-wide property
derby.database.defaultConnectionMode=noAccess
```

Default

fullAccess

Dynamic or static

Dynamic. Current connections are not affected, but all future connections are affected. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.forceDatabaseLock

The *derby.database.forceDatabaseLock* property, if set to true on some platforms, prevents Derby from booting a database if a *db.lck* file is present in the database directory.

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

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

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

Default

False.

Example

```
derby.database.forceDatabaseLock=true
```

Dynamic or static

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

derby.database.fullAccessUsers

The *derby.database.fullAccessUsers* property, one of the user authorization properties, specifies a list of users to which full (read-write) access to a database is granted.

The list consists of user names separated by commas. Do not put spaces after the commas.

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

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

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

For more information about user authorization, see "Configuring user authorization" in the *Derby Security Guide*.

Syntax

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers',
    'commaSeparatedlistOfUsers')
```

Example

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'dba,fred,peter')
--system-level property
derby.database.fullAccessUsers=dba,fred,peter
```

Dynamic or static

Dynamic. Current connections are not affected, but all future connections are affected. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.noAutoBoot

The *derby.database.noAutoBoot* property specifies that a database should not be automatically booted at startup time.

When this property is set to true, this database is booted only on the first connection. Otherwise, the database is booted at startup if the *derby.system.bootAll* property is set to true.

Default

False.

Example

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.noAutoBoot', 'true')
```

Scope

database-wide

Dynamic or static

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

derby.database.propertiesOnly

The *derby.database.propertiesOnly* property, when set to true, ensures that database-wide properties cannot be overridden by system-wide properties.

When this property is set to false, or not set, database-wide properties can be overridden by system-wide properties (see "Precedence of properties" in the *Derby Developer's Guide*).

This property ensures that a database's environment cannot be modified by the environment in which it is booted.

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

Default

False.

Example

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.propertiesOnly','true')
```

Dynamic or static

This property is dynamic; if you change it while Derby is running, the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.readOnlyAccessUsers

The *derby.database.readOnlyAccessUsers* property, one of the user authorization properties, specifies a list of users to which read-only access to a database is granted.

The list consists of user names separated by commas. Do not put spaces after the commas.

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

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

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

Syntax

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readOnlyAccessUsers',
```

```
    'commaSeparatedListOfUsers')
```

Example

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readOnlyAccessUsers', 'ralph,guest')
-- system-level property
derby.database.readOnlyAccessUsers=ralph,guest
```

Dynamic or static

Dynamic. Current connection is not affected, but all future connections are affected. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.sqlAuthorization

The *derby.database.sqlAuthorization* property, one of the user authorization properties, enables the SQL standard authorization mode for the database or system on which this property is set.

The possible values are:

- *TRUE*

  SQL authorization for the database or system is enabled, which allows the use of GRANT and REVOKE statements.
- *FALSE*

  SQL authorization for the database or system is disabled. After this property is set to TRUE, the property cannot be set back to FALSE.

The values are not case-sensitive.

Note: If you set this property as a system property before you create the databases, all new databases will automatically have SQL authorization enabled. If the databases already exist, you can set this property only as a database property.

Derby uses the type of user authentication that is specified with the *derby.authentication.provider* property.

If the *derby.authentication.provider* property specifies NATIVE authentication, Derby behaves as if *derby.database.sqlAuthorization* were set to *TRUE*, regardless of how *derby.database.sqlAuthorization* has been set by other means.

For more information about user authorization, see "Configuring user authorization" in the *Derby Security Guide*.

Example

```
-- system-wide property
derby.database.sqlAuthorization=true
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.sqlAuthorization', 'true');
```

Default

FALSE

Dynamic or static

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

derby.infolog.append

The *derby.infolog.append* property specifies whether to append to or overwrite (delete and recreate) the *derby.log* file when the Derby engine is started.

The *derby.log* file is used to record errors and other information. This information can help you debug problems within a system.

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

See *derby.stream.error.style* for information on how this property works if the *derby.stream.error.style* property is set.

Default

False.

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

Example

```
derby.infolog.append=true
```

Scope

system-wide

Dynamic or static

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

derby.jdbc.xaTransactionTimeout

The *derby.jdbc.xaTransactionTimeout* property specifies the default value of the XA transaction timeout that is used when a user either does not specify the XA transaction timeout or requests to use the default value.

It is possible to use the `XAResource.setTransactionTimeout` method to specify the XA transaction timeout value for the global transaction.

A zero or negative value for this property means that the transaction timeout is not used.

Default

The transaction timeout is not used.

Example

```
-- system-wide property
derby.jdbc.xaTransactionTimeout=120
```

```
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.jdbc.xaTransactionTimeout', '120')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.language.logQueryPlan

The *derby.language.logQueryPlan* property, when set to true, tells Derby to write the query plan information into the `derby.log` file for all executed queries.

This information can help you debug problems within a system.

Example

```
derby.language.logQueryPlan=true
```

Default

False.

Dynamic or static

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

derby.language.logStatementText

The *derby.language.logStatementText* property, when set to true, tells Derby to write the text and parameter values of all executed statements to the information log at the beginning of execution.

It also writes information about commits and rollbacks. Information includes the time and thread number.

This property is useful for debugging.

Example

```
derby.language.logStatementText=true
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.language.logStatementText', 'true')
```

Default

False.

Dynamic or static

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

derby.language.sequence.preallocator

The *derby.language.sequence.preallocator* property specifies how many values to preallocate for sequences.

If the database is shut down in an orderly fashion, Derby will not leak unused preallocated values. Instead, any unused values will be thrown away, and the sequence generator will continue where it left off once the database reboots. However, if the database exits unexpectedly, the sequence generator will skip the unused preallocated values when the database comes up again. This will leave a gap between the last NEXT VALUE FOR (issued before the database exited unexpectedly) and the next NEXT VALUE FOR (issued after the database reboots).

Syntax

```
derby.language.sequence.preallocator=number
```

or

```
derby.language.sequence.preallocator=className
```

If set to a positive number, that is the number of values which Derby preallocates for each sequence. A higher value may improve the concurrency of sequences.

If set to a class name, that class must implement *org.apache.derby.catalog.SequencePreallocator*. The class customizes the size of the preallocation range for each sequence. For more information, see the public API documentation for *org.apache.derby.catalog.SequencePreallocator*.

Default

100

By default, Derby preallocates 100 values for each sequence.

Example

```
derby.language.sequence.preallocator=125
```

Scope

system-wide, database-wide

Dynamic or static

This property is semi-static; changing it while Derby is running will not affect sequences which are already being used. However, the new value will be picked up by sequences which weren't being used before the value was changed. In addition, DDL causes the old value to be forgotten. After performing DDL, the new value will be picked up by all sequences.

derby.language.statementCacheSize

The *derby.language.statementCacheSize* property defines the size, in number of statements, of the database statement cache (prepared statements kept in memory).

This property controls the number of precompiled statements which Derby keeps in its statement cache. Consider raising this number if statement preparation is taking too much time.

For more information on the statement cache, see "Using the statement cache" in *Tuning Derby*.

Default

100 statements.

Example

```
derby.language.statementCacheSize=200
```

Dynamic or static

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

derby.locks.deadlockTimeout

The *derby.locks.deadlockTimeout* property determines the number of seconds after which Derby checks whether a transaction waiting to obtain a lock is involved in a deadlock.

If a deadlock has occurred, and Derby chooses the transaction as a deadlock victim, Derby aborts the transaction. The transaction receives an *SQLException* of *SQLState* 40001. If the transaction is not chosen as the victim, it continues to wait for a lock if *derby.locks.waitTimeout* is set to a higher value than the value of *derby.locks.deadlockTimeout*.

If this property is set to a higher value than *derby.locks.waitTimeout*, no deadlock checking occurs.

For more information about deadlock checking, see "Deadlocks" in the *Derby Developer's Guide*.

Default

20 seconds.

Example

```
derby.locks.deadlockTimeout=30
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.deadlockTimeout', '30')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.locks.deadlockTrace

The *derby.locks.deadlockTrace* property causes a detailed list of locks at the time of a deadlock or a timeout to be written to the error log (typically the file *derby.log*).

For a deadlock, Derby describes the cycle of locks which caused the deadlock. For a timeout, Derby prints the entire lock list at the time of the timeout. This property is meaningful only if the *derby.locks.monitor* property is set to *true*.

Note: This level of debugging is intrusive: it can alter the timing of the application, reduce performance severely, and produce a large error log file. It should be used with care.

Default

False.

Example

```
-- system property
derby.locks.deadlockTrace=true

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.deadlockTrace', 'true')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.locks.escalationThreshold

The *derby.locks.escalationThreshold* property is used by the Derby system at runtime in determining when to attempt to escalate locking for at least one of the tables involved in a transaction from row-level locking to table-level locking.

A large number of row locks use a lot of resources. If nearly all the rows are locked, it might be worth the slight decrease in concurrency to lock the entire table to avoid the large number of row locks.

For more information, see "Locking and performance" in *Tuning Derby*.

It is useful to increase this value for large systems (such as enterprise-level servers, where there is more than 64 MB of memory), and to decrease it for very small systems (such as palmtops).

Syntax

```
derby.locks.escalationThreshold=numberOfLocks
```

Default

5000.

Minimum value

100.

Maximum value

2,147,483,647.

Example

```
-- system-wide property
derby.locks.escalationThreshold=1000
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.escalationThreshold',
    '1000')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.locks.monitor

The *derby.locks.monitor* property specifies that all deadlock errors are logged to the error log.

If *derby.stream.error.logSeverityLevel* is set to ignore deadlock errors, *derby.locks.monitor* overrides it.

Default

False.

Example

```
-- system property
derby.locks.monitor=true

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.monitor', 'true')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.locks.waitTimeout

The *derby.locks.waitTimeout* property specifies the number of seconds after which Derby aborts a transaction when it is waiting for a lock.

When Derby aborts (and rolls back) the transaction, the transaction receives an *SQLException* of *SQLState* 40XL1.

The time specified by this property is approximate.

A zero value for this property means that Derby aborts a transaction any time it cannot immediately obtain a lock that it requests.

A negative value for this property is equivalent to an infinite wait time; the transaction waits forever to obtain the lock.

If this property is set to a value greater than or equal to zero but less than the value of *derby.locks.deadlockTimeout*, Derby never performs any deadlock checking.

Default

60 seconds.

Example

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.waitTimeout', '15')
derby.locks.waitTimeout=60
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.replication.logBufferSize

The *derby.replication.logBufferSize* property specifies the size of the replication log buffers in bytes.

These buffers store the log on the master side before it is shipped to the slave. There is a total of 10 such buffers.

Large buffers increase memory usage but reduce the chance that the buffers will fill up (in turn increasing response time for transactions on the master, as described in the failure situation "The master Derby instance is not able to send log data to the slave at the same pace as the log is generated" in the topic "Replication failure handling" in the *Derby Server and Administration Guide*).

You can also use the properties *derby.replication.minLogShippingInterval* and *derby.replication.maxLogShippingInterval* to tune the rate at which the log is shipped from the master to the slave.

Minimum value

8192 (8 KB).

Maximum value

The maximum value is 1048576 (1 MB).

Default

32768 bytes (32KB).

Example

```
derby.replication.logBufferSize=65536
```

Scope

system-wide

Dynamic or static

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

derby.replication.maxLogShippingInterval

The *derby.replication.maxLogShippingInterval* property specifies, in milliseconds, the longest interval between two consecutive shipments of the transaction log from the master to the slave.

This property provides a "soft" guarantee that the slave will not deviate more than this number of milliseconds from the master.

The value specified for the *derby.replication.maxLogShippingInterval* property must be at least ten times the value specified for the *derby.replication.minLogShippingInterval* property. If you set *derby.replication.maxLogShippingInterval* to a lower value, Derby changes the *derby.replication.minLogShippingInterval* property value to the value of the *derby.replication.maxLogShippingInterval* property divided by 10.

Default

5000 milliseconds (5 seconds).

Example

```
derby.replication.maxLogShippingInterval=10000
```

Scope

system-wide

Dynamic or static

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

derby.replication.minLogShippingInterval

The *derby.replication.minLogShippingInterval* property specifies, in milliseconds, the shortest interval between two consecutive shipments of the transaction log from the master to the slave.

The value specified for the *derby.replication.minLogShippingInterval* property must be no more than one-tenth the value specified for the *derby.replication.maxLogShippingInterval* property. If you set *derby.replication.minLogShippingInterval* to a higher value, Derby changes the *derby.replication.minLogShippingInterval* property value to the value of the *derby.replication.maxLogShippingInterval* property divided by 10.

Default

100 milliseconds.

Example

```
derby.replication.minLogShippingInterval=500
```

Scope

system-wide

Dynamic or static

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

derby.replication.verbose

The *derby.replication.verbose* property specifies whether replication messages are written to the Derby log.

Default

True.

Example

```
derby.replication.verbose=false
```

Scope

system-wide

Dynamic or static

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

derby.storage.indexStats.auto

The *derby.storage.indexStats.auto* property, if set to true (the default), specifies that outdated index cardinality statistics are updated automatically during query compilation.

The query compiler schedules a job that updates the statistics in a separate thread.

The thread that updates the statistics in the background may affect the performance of the user threads. If this causes problems, you can set *derby.storage.indexStats.auto* to false.

To diagnose problems with automatic updating of index statistics, an application can set the property *derby.storage.indexStats.log* to true. The *derby.storage.indexStats.trace* property can be used to provide more detailed information.

For more information about index statistics, see "Working with cardinality statistics" in *Tuning Derby*.

Default

True.

Example

```
-- system-wide property
derby.storage.indexStats.auto=false
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.indexStats.auto', 'false')
```

Dynamic or static

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

derby.storage.indexStats.log

The *derby.storage.indexStats.log* property, if set to true, specifies that the thread that updates index cardinality statistics during query compilation will write messages to the Derby system log (*derby.log*) every time it performs a task.

The log entries should help you to diagnose problems with the automatic updating of index statistics.

The *derby.storage.indexStats.trace* property can be used to provide more detailed information and to specify where the trace output should appear.

To disable the automatic updating of index statistics, set the database property *derby.storage.indexStats.auto* to false.

For more information about index statistics, see "Working with cardinality statistics" in *Tuning Derby*.

Default

False.

Example

```
-- system-wide property
derby.storage.indexStats.log=true
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.indexStats.log', 'true')
```

Dynamic or static

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

derby.storage.indexStats.trace

The *derby.storage.indexStats.trace* property allows you to enable more detailed logging of index cardinality statistics than the *derby.storage.indexStats.log* property provides.

This property also allows you to specify where the tracing output should appear. Valid values are as follows:

- `off`

  Tracing is disabled (the default).
- `log`

  Tracing output goes to the log file, *derby.log*.
- `stdout`

  Tracing output goes to standard output.
- `both`

  Tracing output goes to both *derby.log* and standard output.

See *derby.storage.indexStats.log* for details on that property. For more information about index statistics, see "Working with cardinality statistics" in *Tuning Derby*.

Syntax

```
derby.storage.indexStats.trace=
    { off | log | stdout | both }
```

Default

`off`.

Example

```
-- system-wide property
```

```
derby.storage.indexStats.trace=log
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.indexStats.trace', 'stdout')
```

Dynamic or static

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

derby.storage.initialPages

The *derby.storage.initialPages* property creates a Derby table or index with a number of pages already pre-allocated.

The on-disk size of a Derby table grows by one page at a time until eight pages of user data (or nine pages of total disk use; one is used for overhead) have been allocated. Then it will grow by eight pages at a time if possible.

To pre-allocate pages, specify the property prior to the CREATE TABLE or CREATE INDEX statement. The property value defines the number of user pages the table or index is to be created with. The purpose of this property is to preallocate a table or index of reasonable size if the user expects that a large amount of data will be inserted into the table or index. A table or index that has the pre-allocated pages will enjoy a small performance improvement over a table or index that has no pre-allocated pages when the data are loaded.

The total desired size of the table or index should be the following number of bytes:

```
(1 + derby.storage.initialPages) * derby.storage.pageSize
```

When you create a table or an index after setting this property, Derby attempts to preallocate the requested number of user pages. However, the operations do not fail even if they are unable to preallocate the requested number of pages, as long as they allocate at least one page.

Default

1 page.

Minimum value

The minimum number of *initialPages* is 1.

Maximum value

The maximum number of *initialPages* is 1000.

Example

```
-- system-wide property
derby.storage.initialPages=30

-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
'derby.storage.initialPages', '30')
```

Dynamic or static

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

derby.storage.minimumRecordSize

The *derby.storage.minimumRecordSize* property indicates the minimum user row size in bytes for on-disk database pages for tables when you are creating a table.

This property ensures that there is enough room for a row to grow on a page when updated without having to overflow. This is generally most useful for VARCHAR and VARCHAR FOR BIT DATA data types and for tables that are updated often, in which the rows start small and grow due to updates. Reserving the space at the time of insertion minimizes row overflow due to updates, but it can result in wasted space. Set the property prior to issuing the CREATE TABLE statement.

See also *derby.storage.pageReservedSpace*.

Valid conglomerates

Tables only.

Default

12 bytes.

Minimum value

12 bytes.

Maximum value

derby.storage.pageSize * (1 - *derby.storage.pageReservedSpace*/100) " 100.

If you set this property to a value outside the legal range, Derby uses the default value.

Example

```
-- changing the default for the system
derby.storage.minimumRecordSize=128
-- changing the default for the database
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.minimumRecordSize',
    '128')
```

Dynamic or static

This property is dynamic; if you change it while Derby is running, the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.storage.pageCacheSize

The *derby.storage.pageCacheSize* property defines the size, in number of pages, of the data page cache in the database (data pages kept in memory).

The actual amount of memory the page cache will use depends on the following:

- The size of the cache, configured with the *derby.storage.pageCacheSize* property.
- The size of the pages, configured with the *derby.storage.pageSize* property. Derby automatically tunes for the database page size. If you have long columns, the default page size for the table is set to 32768 bytes. Otherwise, the default is 4096 bytes.
- Overhead, which varies with JVMs.

When increasing the size of the page cache, you typically have to allow more memory for the Java heap when starting the embedding application (taking into consideration, of course, the memory needs of the embedding application as well). For example, using the

default page size of 4K, a page cache size of 2000 pages will require at least 8 MB of memory (and probably more, given the overhead).

The minimum value is 40 pages. If you specify a lower value, Derby uses the default value.

Default

1000 pages.

Example

```
derby.storage.pageCacheSize=160
```

Dynamic or static

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

derby.storage.pageReservedSpace

The *derby.storage.pageReservedSpace* property defines the percentage of space reserved for updates on an on-disk database page for tables only (not indexes) and indicates the percentage of space to keep free on a page when inserting.

Leaving reserved space on a page can minimize row overflow (and the associated performance hit) during updates. Once a page has been filled up to the reserved-space threshold, no new rows are allowed on the page. This reserved space is used only for rows that increase in size when updated, not for new inserts. Set this property prior to issuing the CREATE TABLE statement.

Regardless of the value of *derby.storage.pageReservedSpace*, an empty page always accepts at least one row.

Valid conglomerates

Tables only.

Default

20%.

Minimum value

The minimum value is 0% and the maximum is 100%. If you specify a value outside this range, Derby uses the default value of 20%.

Example

```
-- modifying the default for the system
derby.storage.pageReservedSpace=40
-- modifying the default for the database
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.pageReservedSpace',
    '40')
```

Dynamic or static

This property is dynamic: if you change it while Derby is running, the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.storage.pageSize

The *derby.storage.pageSize* property defines the page size, in bytes, for on-disk database pages for tables or indexes used during table or index creation.

Set this property prior to issuing the CREATE TABLE or CREATE INDEX statement. This value will be used for the lifetime of the newly created conglomerates.

Valid conglomerates

Tables and indexes, including the indexes created to enforce constraints.

Default

Derby automatically tunes for the database page size. If you have any LONG VARCHAR, LONG VARCHAR FOR BIT DATA, BLOB, or CLOB columns, or if Derby estimates that the total length of the columns declared at create time is greater than 4096 bytes, the default page size for the table is set to 32768 bytes. Otherwise, the default is 4096 bytes.

Valid values

Page size can only be one of the following values: 4096, 8192, 16384, or 32768 bytes. If you specify an invalid value, Derby uses the default value.

Example

```
-- changing the default for the system
derby.storage.pageSize=8192
-- changing the default for the database
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.pageSize',
    '8192')
```

Dynamic or static

This property is dynamic; if you change it while Derby is running, the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.storage.rowLocking

The *derby.storage.rowLocking* property, if set to true, enables row-level locking.

When you disable row-level locking, you use table-level locking.

Row-level locking uses more system resources but allows greater concurrency, which works better in multi-user systems. Table-level locking works best with single-user applications or read-only applications.

If you use row-level locking (the default), the system decides whether to use table-level locking or row-level locking for each table in each DML statement. In certain situations, the system might choose to escalate the locking scheme from row-level locking to table-level locking to improve performance.

For more information about locking, see "Locking and performance" in *Tuning Derby*, and "Locking, concurrency, and isolation" in the *Derby Developer's Guide*.

Default

True.

Example

```
-- system-wide property
derby.storage.rowLocking=false
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
```

```
    'derby.storage.rowLocking', 'false')
```

Dynamic or static

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

derby.storage.tempDirectory

The *derby.storage.tempDirectory* property defines the location on disk for temporary file space needed by Derby for performing large sorts and deferred deletes and updates.

Temporary files are automatically deleted after use and are removed when the database restarts after a crash. The temporary directory named by this property will be created if it does not exist, but will not be deleted when the system shuts down. The path name specified by this property must have file separators that are appropriate to the current operating system.

This property allows databases located on read-only media to write temporary files to a writable location. If this property is not set, databases located on read-only media might get an error like the following:

```
ERROR XSDF1: Exception during creation
 of file  c:\databases\db\tmp\T887256591756.tmp
for container
ERROR XJ001: Java exception:
'a:\databases\db\tmp\T887256591756.tmp: java.io.IOException'.
```

This property moves the temporary directories for all databases being used by the Derby system. Derby makes temporary directories for each database under the directory referenced by this property. For example, if the property is set as follows:

```
derby.storage.tempDirectory=C:/Temp/dbtemp
```

the temporary directories for the databases in *C:\databases\db1* and *C:\databases\db2* will be in *C:\Temp\dbtemp\db1* and *C:\Temp\dbtemp\db2*, respectively.

The temporary files of two databases running concurrently with the same name (for example, *C:\databases\db1* and *E:\databases\db1*) will conflict with each other if the *derby.storage.tempDirectory* property is set. This will cause incorrect results, so users are advised to give databases unique names.

Default

A subdirectory named *tmp* under the database directory.

For example, if the database *db1* is stored in *C:\databases\db1*, the temporary files are created in *C:\databases\db1\tmp*.

Example

```
-- system-wide property
derby.storage.tempDirectory=c:/Temp/dbtemp
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.tempDirectory',
    'c:/Temp/dbtemp')
```

Dynamic or static

This property is static; you must restart Derby for a change to take effect.

derby.storage.useDefaultFilePermissions

The *derby.storage.useDefaultFilePermissions* property overrides the default access to files on systems that run on the Java SE 7 platform.

If you run with Java SE 7 or later, and if you start the Derby Network Server from the command line, access to databases and to other files created by Derby is by default restricted to the operating system account that started the Network Server. File access is not restricted for embedded databases or for databases managed by servers that are started programmatically inside application code using the Derby API.

You can override this default behavior by setting the system property *derby.storage.useDefaultFilePermissions* to either true or false.

If you run with the Java SE 6 platform, this property is ignored, and Derby uses the default file permissions the user has set for their system.

The two tables that follow show how file access works with Java SE 6 and with Java SE 7 and later JVMs. In both tables,

- "Environment" means that access is controlled entirely by the JVM environment and the file location only (that is, by the umask setting on UNIX and Linux systems and by the default file permissions on Windows NTFS).
- "Restricted" means that Derby restricts access to the operating system account that started the JVM.

The following table shows how file access works on Java SE 6 systems.

Table 151. File access on Java SE 6 systems

Property Setting	Server Started from Command Line	Server Started Programmatically or Embedded
Not applicable	Environment	Environment

The following table shows how file access works on Java SE 7 and later systems with various settings of the `derby.storage.useDefaultFilePermissions` property.

Table 152. File access on Java SE 7 and later systems

Property Setting	Server Started from Command Line	Server Started Programmatically or Embedded
No property specified	Restricted	Environment
Property set to true	Environment	Environment
Property set to false	Restricted	Restricted

For more information, see "Restricting file permissions" in the *Derby Security Guide*.

Default

By default, this property is not set.

Example

```
derby.storage.useDefaultFilePermissions=true
```

Scope

system-wide

Dynamic or static

Dynamic. Existing files will keep their previous permissions, but files created after the property is set will have the permissions specified by the property. If you want all the files in the database to have the same permissions, do not change the property while Derby is running.

For information about dynamic changes to properties, see Dynamic and static properties.

derby.stream.error.extendedDiagSeverityLevel

The *derby.stream.error.extendedDiagSeverityLevel* property specifies whether thread dump information and extended diagnostic information are created, and at what level, in the event of a system crash or session error.

If errors have a severity level greater than or equal to the value of the *derby.stream.error.extendedDiagSeverityLevel* property, thread dump and diagnostic information will appear in the *derby.log* file. In addition, with IBM Java Virtual Machines (JVMs), a javacore file with additional information is created.

To allow the information to be dumped to the log, you must grant two permissions to Derby in your security policy file. See "Configuring Java security" in the *Derby Security Guide*.

Any error raised in a Derby system is given a level of severity. This property indicates the minimum severity necessary for an error to appear in the log file. The severities are defined in the class *org.apache.derby.types.ExceptionSeverity*. The higher the number, the more severe the error.

- *10000*

  Warnings.
- *20000*

  Errors that cause the statement to be rolled back, such as syntax errors and constraint violations.
- *30000*

  Errors that cause the transaction to be rolled back, such as deadlocks.
- *40000*

  Errors that cause the session to be closed.
- *45000*

  Errors that cause the database to be closed.
- *50000*

  Errors that shut down the Derby system.

Default

40000.

Example

```
// send errors of level 30000 and higher to the log
derby.stream.error.extendedDiagSeverityLevel=30000
```

Scope

system-wide

Dynamic or static

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

derby.stream.error.field

The *derby.stream.error.field* property specifies a static field that references a stream to which the error log is written.

The field is specified using the fully qualified name of the class, then a dot (.), and then the field name. The field must be public and static. Its type can be either *java.io.OutputStream* or *java.io.Writer*.

The field is accessed once at Derby boot time, and the value is used until Derby is rebooted. If the field is null, the error stream defaults to the system error stream (*java.lang.System.err*).

If the field does not exist or is inaccessible, the error stream defaults to the system error stream. Derby will not call the *close()* method of the object obtained from the field.

If you specify this property, the property setting appears in the error log.

Default

None.

Example

```
derby.stream.error.field=java.lang.System.err
```

Scope

system-wide

Dynamic or static

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

derby.stream.error.file

The *derby.stream.error.file* property specifies the name of the file to which the error log is written.

If the file name is relative, it is taken as relative to the system directory.

If this property is set, the *derby.stream.error.method* and *derby.stream.error.field* properties are ignored.

If you specify this property, the property setting appears in the error log.

Default

derby.log.

Example

```
derby.stream.error.file=error.txt
```

Dynamic or static

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

derby.stream.error.logBootTrace

The *derby.stream.error.logBootTrace* property specifies whether boot errors are written to the Derby log file.

This property helps in diagnosing double-booting problems. Typically, when two Java Virtual Machines (JVMs) or class loaders attempt to boot Derby, an error message like the following appears:

```
ERROR XJ040: Failed to start database 'testdb' with class loader
sun.misc.Launcher$AppClassLoader@481e481e, see the next exception for
details.
ERROR XSDB6: Another instance of Derby may have already booted the
database C:\derby\testdb.
```

The message will also show the stack trace and class loader of the failed boot attempt. It is sometimes also useful to see the stack trace when the first successful boot attempt occurred. To see the stack trace of successful boots and shutdowns, set *derby.stream.error.logBootTrace=true* to trace the successful attempt. If you think that both attempts should be from the same class loader context, check also the class loader information for boot and shutdown attempts and make sure they all come from the same class loader context.

For more information, see "Double-booting system behavior" in *Derby Developer's Guide*.

Default

False.

Example

```
derby.stream.error.logBootTrace=true
```

Scope

system-wide

Dynamic or static

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

derby.stream.error.logSeverityLevel

The *derby.stream.error.logSeverityLevel* property specifies which errors are logged to the Derby error log (typically the *derby.log* file).

In test environments, use the setting *derby.stream.error.logSeverityLevel=0* so that all problems are reported.

Any error raised in a Derby system is given a level of severity. This property indicates the minimum severity necessary for an error to appear in the error log. The severities are defined in the class *org.apache.derby.types.ExceptionSeverity*. The higher the number, the more severe the error.

- *20000*

  Errors that cause the statement to be rolled back, for example syntax errors and constraint violations.
- *30000*

  Errors that cause the transaction to be rolled back, for example deadlocks.
- *40000*

  Errors that cause the connection to be closed.
- *50000*

Errors that shut down the Derby system.

Default

40000.

Example

```
// send errors of level 30000 and higher to the log
derby.stream.error.logSeverityLevel=30000
```

Dynamic or static

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

derby.stream.error.method

The *derby.stream.error.method* property specifies a static method that returns a stream to which the Derby error log is written.

Specify the method using the fully qualified name of the class, then a dot (.) and then the method name. The method must be public and static. Its return type can be either *java.io.OutputStream* or *java.io.Writer*. Derby will not call the *close()* method of the object returned by the method.

The method is called once at Derby boot time, and the return value is used for the lifetime of Derby. If the method returns null, the error stream defaults to the system error stream. If the method does not exist or is inaccessible, the error stream defaults to the system error stream (*java.lang.System.err*).

If the value of this property is set, the property *derby.stream.error.field* is ignored.

If you specify this property, the property setting appears in the error log.

Default

Not set.

Example

```
derby.stream.error.method=java.sql.DriverManager.getLogStream
```

Scope

system-wide

Dynamic or static

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

derby.stream.error.rollingFile.count

The *derby.stream.error.rollingFile.count* property specifies the number of rolling log files to permit before deleting the oldest file when rolling to the next file, if *derby.stream.error.style* is set to `rollingFile`.

If *derby.stream.error.style* is not set, this property setting is ignored.

You can override other rolling log file defaults by setting *derby.stream.error.rollingFile.limit* or *derby.stream.error.rollingFile.pattern*.

Syntax

```
derby.stream.error.rollingFile.count=count
```

The specified *count* value must be equal to or greater than 1.

Default

10.

Example

```
derby.stream.error.rollingFile.count=1
```

Scope

system-wide

Dynamic or static

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

derby.stream.error.rollingFile.limit

The *derby.stream.error.rollingFile.limit* property specifies the number of bytes to which to limit each rolling log file before it is rolled over to the next file, if *derby.stream.error.style* is set to `rollingFile`.

If *derby.stream.error.style* is not set, this property setting is ignored.

You can override other rolling log file defaults by setting *derby.stream.error.rollingFile.count* or *derby.stream.error.rollingFile.pattern*.

Syntax

```
derby.stream.error.rollingFile.limit=limit
```

The specified *limit* value must be equal to or greater than 0 (zero). If you specify 0 as the limit, the file size has no limit, and the log file will never roll over.

Default

1024000 (one megabyte).

Example

```
derby.stream.error.rollingFile.limit=2048000
```

Scope

system-wide

Dynamic or static

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

derby.stream.error.rollingFile.pattern

The *derby.stream.error.rollingFile.pattern* property specifies the naming pattern to use for the rolling log files, if *derby.stream.error.style* is set to `rollingFile`.

If *derby.stream.error.style* is not set, this property setting is ignored.

You can override other rolling log file defaults by setting *derby.stream.error.rollingFile.count* or *derby.stream.error.rollingFile.limit*.

A pattern consists of a string that includes the following special components that will be replaced at runtime.

Table 153. Rolling log file pattern components

Pattern component	Description
/	Local path name separator
%t	The system temporary directory
%h	The value of the *user.home* system property
%d	The value of the *derby.system.home* system property
%g	The generation number to distinguish rotated log files. Generation numbers follow the sequence 0, 1, 2, and so on. If you do not specify a %g field and the file count is greater than one, the generation number will be added to the end of the generated filename, after a dot.
%u	A unique number to resolve conflicts.
%%	Translates to a single percent sign (%)

Thus, for example, a *derby.stream.error.rollingFile.pattern* setting of `%t/java%g.log` with a *derby.stream.error.rollingFile.count* setting of 2 might cause log files on a UNIX system to be written to */var/tmp/java0.log* and */var/tmp/java1.log*, whereas on a Windows 7 system they might be written to *%USERPROFILE%\AppData\Local\Temp\java0.log* and *%USERPROFILE%\AppData\Local\Temp\java1.log*.

Normally, the %u unique field is set to 0 (zero). However, if Derby tries to open the file by the specified name and finds that the file is currently in use by another process, it will increment the unique number field and try again. This action will be repeated until Derby finds a file name that is not currently in use. If there is a conflict and no %u field has been specified, the unique number will be added at the end of the filename after a dot. (This will be after any automatically added generation number.)

For example, if three processes are all trying to log to `fred%u.%g.txt`, they might have log files named *fred0.0.txt*, *fred1.0.txt*, *fred2.0.txt* as the first file in their rotating sequences.

Note: The use of unique fields to avoid conflicts is guaranteed to work reliably only when you are using a local disk file system.

Syntax

```
derby.stream.error.rollingFile.pattern=pattern
```

The number of characters in the specified *pattern* value must be equal to or greater than 1.

Default

`%d/derby-%g.log`.

Example

The following setting creates files named *myDBlog-0.txt*, *myDBlog-1.txt*, and so on, in the user's home directory:

```
derby.stream.error.rollingFile.pattern=%h/myDBlog-%g.txt
```

Scope

system-wide

Dynamic or static

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

derby.stream.error.style

The *derby.stream.error.style* property specifies that the Derby log file should be rolled over when it reaches a certain size.

Function

If you set this property to `rollingFile` (the only supported value), Derby by default creates up to 10 rolling files named *derby-0.log*, *derby-1.log*, and so on, up to *derby-9.log*, each with a maximum length of 1,024,000 bytes.

You can override any of these defaults by setting one or more of the following properties:

- *derby.stream.error.rollingFile.count*
- *derby.stream.error.rollingFile.limit*
- *derby.stream.error.rollingFile.pattern*

If you set this property, the *derby.stream.error.field*, *derby.stream.error.file*, and *derby.stream.error.method* properties are ignored.

This property works in accordance with the setting of the *derby.infolog.append* property. If *derby.infolog.append* is not set or is set to false, any existing log files will be rolled over and a new log file will be created when the Derby engine is started. If *derby.infolog.append* is set to true, the latest existing log file, if any, will be appended to.

Syntax

```
derby.stream.error.style=style
```

Default

Not set.

Example

```
derby.stream.error.style=rollingFile
```

Scope

system-wide

Dynamic or static

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

derby.system.bootAll

The *derby.system.bootAll* property specifies that all databases in the directory specified by the *derby.system.home* property should be automatically booted at startup time.

When this property is set to true, databases in the *derby.system.home* directory are booted at startup. Otherwise, databases are booted when you first connect to them.

You may want to use the *derby.system.bootAll* property to avoid a delay at first connection time. After a crash, a boot that requires recovery can take a long time, and you may want to perform this boot as soon as Derby is restarted.

You can set the *derby.database.noAutoBoot* property on a particular database if you want to prevent it from being automatically booted at startup.

Default

False.

Scope

system-wide

Example

```
derby.system.bootAll=true
```

Dynamic or static

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

derby.system.durability

The *derby.system.durability* property changes the default durability of Derby to improve performance at the expense of consistency and durability of the database.

The only valid supported case insensitive value is `test`. If this property is set to any value other than `test`, this property setting is ignored. When *derby.system.durability* is set to `test`, the store system will not force I/O synchronization calls for:

- The log file at each commit
- The log file before a data page is forced to disk
- Page allocation when a file is grown
- Data writes during checkpoints

While performance is improved, note that under these conditions, a commit no longer guarantees that the transaction's modification will survive a system crash or JVM termination, the database may not recover successfully upon restart, a near-full disk at runtime may cause unexpected errors, and the database may be in an inconsistent state.

If you boot the database with this property set to `test`, the following warning message is logged in the `derby.log` file:

```
WARNING: The database is booted with derby.system.durability=test.
In this mode, it is possible that database may not be able
to recover, committed transactions may be lost, and the database
may be in an inconsistent state. Please use this mode only when
these consequences are acceptable.
```

A similar message will appear in the `derby.log` file if the database was booted with `derby.system.durability=test` at any time previously.

Once the database is booted with `derby.system.durability=test`, there are no guarantees that the database is consistent.

Default

This property is ignored by default.

Supported values

The only supported value is `test`.

Example

```
derby.system.durability=test
```

Since this is a system property, you can set it in the `derby.properties` file or on the command line of the JVM when starting the application.

You might enable this property when using Derby as a test database where consistency or recoverability is not an issue.

Dynamic or static

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

derby.system.home

The *derby.system.home* property specifies the Derby system directory, which is the directory that contains subdirectories holding databases that you create and the text file *derby.properties*.

If the system directory that you specify with *derby.system.home* does not exist at startup, Derby creates the directory automatically.

Default

Current directory (the value of the JVM system property *user.dir*).

If you do not explicitly set the *derby.system.home* property when starting Derby, the default is the directory in which Derby was started.

Note: You should always explicitly set the value of *derby.system.home*.

Example

```
-Dderby.system.home=C:\derby
```

Dynamic or static

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

derby.user.UserName

The *derby.user.UserName* property caches user DNs locally when *derby.authentication.provider* is set to LDAP and *derby.authentication.ldap.searchFilter* is set to `derby.user`.

See *derby.authentication.provider* and *derby.authentication.ldap.searchFilter* for more information.

When you provide a user DN with this property, Derby is able to avoid an LDAP search for that user's DN before authenticating. For those users without DNs defined with this property, Derby performs a search using the default value of *derby.authentication.ldap.searchFilter*.

User names are *SQLIdentifier*s and can be delimited.

Syntax

```
derby.user.UserName=userDN
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.UserName',
    'userDN')
```

Default

None.

Example

```
-- system-level property
derby.user.Diana=uid=Diana,ou=People,o=example.com

-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.richard',
    'uid=richard, ou=People, o=example.com')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

DataDictionaryVersion

The *DataDictionaryVersion* property shows the version of the on-disk data in the database.

The version is the first two numbers (the major and minor release values) in a Derby release identifier. For newly created databases and for soft-upgraded databases, this is the release identifier of the Derby engine jar used to create the database. For fully upgraded databases, this is the release identifier of the Derby engine jar used to perform a full upgrade of the database. See *upgrade=true* attribute for more information about upgrading databases.

You can retrieve the value of this property by using the SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY system function, but you cannot change it.

Syntax

```
-- database-level property
VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('DataDictionaryVersion')
```

Example

```
ij> connect 'jdbc:derby:firstdb';
ij> VALUES
 SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('DataDictionaryVersion');
1

------------------------------------------------------------------------
10.10


1 row selected
```

Java EE compliance: Java Transaction API and javax.sql interfaces

The Java Platform, Enterprise Edition (the Java EE platform) 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 Platform, Standard Edition (the Java SE platform), the Java EE platform adds support for Enterprise JavaBeans (EJB) technology, the Java Persistence API, JavaServer Faces technology, Java Servlet technology, JavaServer Pages (JSP) technology, and many more. The Java EE platform architecture is used to bring together existing technologies and enterprise applications in a single, manageable environment.

Derby is a Java EE platform conformant component in a distributed Java EE system. As such, Derby 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 the Java EE platform, see http://www.oracle.com/technetwork/java/javaee/documentation/index.html.

In order to qualify as a resource manager in a Java EE system, the Java EE platform 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 interfaces, *javax.sql.DataSource*, provides this support (except for the DataSource implementations that support Java SE 8 Compact Profile 2).
- 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 interfaces, *javax.sql.ConnectionPoolDataSource* and *javax.sql.PooledConnection*, provides 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 the *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*), provides 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 on how to use Derby as a resource manager, see "Using Derby as a Java EE Resource Manager" in the *Derby Developer's Guide*.

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.

The implemented APIs are as follows. The following sections provide some implementation details.

- *javax.transaction.xa.XAResource*
- *javax.transaction.xa.Xid*
- *javax.transaction.xa.XAException*

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

The behavior of transactions created by *XAConnection* objects varies according to whether a user name and password are specified when the *XAConnection* object is created.

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.

XA transactions and deferred constraints

Derby defines how some *XAResource* methods behave in conjunction with deferred constraints.

If an application calls *XAResource.prepare(Xid)*, any constraints with a constraint mode of DEFERRED are checked. If there is a violation, Derby throws *XAException.XA_RBINTEGRITY*, and the XA transaction is rolled back.

If an application calls *XAResource.commit(Xid, true)* (with *true* indicating a one-phase commit), any constraints with a constraint mode of DEFERRED are checked. If there is a violation, Derby throws *XAException.XA_RBINTEGRITY*, and the XA transaction is rolled back.

See CONSTRAINT clause and SET CONSTRAINTS statement for more information about deferrable constraints.

javax.sql: JDBC interfaces

Derby implements a number of JDBC interfaces for Java EE compliance.

For more details about these interfaces, see the API documentation for your version of the Java Development Kit, which you can find at http://docs.oracle.com/javase/.

- *javax.sql.DataSource*

  An interface that is a factory for connections to the physical data source that the object represents. An object that implements the DataSource interface will typically be registered with a naming service based on the Java Naming and Directory (JNDI) API. The *org.apache.derby.jdbc* DataSource classes support the JNDI API, with the exception of the DataSource classes that support Java SE 8 Compact Profile 2. See DataSource classes and JDBC support for Java SE 8 Compact Profiles for more information.
- *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 the *ConnectionPoolDataSource* and *PooledConnection* interfaces 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*. This type of connection 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 documentation of API classes and interfaces in the *javadoc* subdirectory.

This section provides a brief overview of the API. Derby does not provide the API documentation 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 the JDBC API, see JDBC reference.

This section 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 on the command line. 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 package *org.apache.derby.tools*.

For information about these classes, see the *Derby Tools and Utilities Guide*.

- *org.apache.derby.tools.ij*

  An SQL scripting tool that can run as an embedded or a remote client/server application.
- *org.apache.derby.tools.sysinfo*

  A command-line, server-side utility that displays information about your Java Virtual Machine (JVM) and Derby product.
- *org.apache.derby.tools.dblook*

  A utility to view all or parts of the Data Definition Language (DDL) for a given database.
- *org.apache.derby.tools.SignatureChecker*

  A utility that identifies any SQL functions and procedures in a database that do not follow the SQL Standard argument matching rules described in Argument matching.

Derby also supports some optional tools that are described in the "Optional tools" section of the *Derby Tools and Utilities Guide*.

JDBC implementation classes

JDBC drivers

Derby has two JDBC drivers.

- *org.apache.derby.jdbc.EmbeddedDriver*

  Used to boot the embedded built-in JDBC driver and the Derby system. See the *Derby Developer's Guide* for more information.
- *org.apache.derby.jdbc.ClientDriver*

  Used to connect to the Derby Network Server in client-server mode. See the *Derby Server and Administration Guide* for more information.

DataSource classes

These classes are all related to Derby's implementation of *javax.sql.DataSource* and related APIs.

For more information, see JDBC reference, JDBC support for Java SE 8 Compact Profiles, "Classes that pertain to resource managers" in the *Derby Developer's Guide*, and "Accessing the Network Server by using a DataSource object" in the *Derby Server and Administration Guide*.

Embedded environment, for applications using Java SE 8 Compact Profile 2:

- *org.apache.derby.jdbc.BasicEmbeddedDataSource40*
- *org.apache.derby.jdbc.BasicEmbeddedConnectionPoolDataSource40*
- *org.apache.derby.jdbc.BasicEmbeddedXADataSource40*

Embedded environment, for applications using all other Java SE versions:

- *org.apache.derby.jdbc.EmbeddedDataSource*
- *org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource*
- *org.apache.derby.jdbc.EmbeddedXADataSource*

Client-server environment, for applications using Java SE 8 Compact Profile 2:

- *org.apache.derby.jdbc.BasicClientDataSource40*
- *org.apache.derby.jdbc.BasicClientConnectionPoolDataSource40*
- *org.apache.derby.jdbc.BasicClientXADataSource40*

Client-server environment, for applications using all other Java SE versions:

- *org.apache.derby.jdbc.ClientDataSource*
- *org.apache.derby.jdbc.ClientConnectionPoolDataSource*
- *org.apache.derby.jdbc.ClientXADataSource*

Applications using Java SE 8 Compact Profile 2 must use the DataSource classes whose names begin with "Basic". Applications using Java SE 8 Compact Profile 3 can use the classes whose names begin with "Basic" if the application does not use the JNDI API, but should normally use the ordinary DataSource classes.

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 more information, see "Specifying authentication with a user-defined class" in the *Derby Security Guide*.

Supported locales

The following table lists the locales supported by Derby.

Use the *territory=ll_CC* URL attribute to set the Derby locale.

Table 154. Supported locales

Locale	*territory=ll_CC* Setting
Chinese (Simplified)	zh_CN
Chinese (Traditional)	zh_TW
Czech	cs
French	fr
German	de_DE
Hungarian	hu
Italian	it
Japanese	ja_JP
Korean	ko_KR
Polish	pl
Portuguese (Brazilian)	pt_BR
Russian	ru
Spanish	es

Derby limitations

This section lists the limitations associated with Derby.

Limitations for database values

The following table lists limitations on various database values in Derby.

Table 155. Database limitations

Value	Limit
Maximum columns in a table	1,012
Maximum columns in a view	5,000
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

DATE, TIME, and TIMESTAMP limitations

The following table lists limitations on date, time, and timestamp values in Derby.

Table 156. DATE, TIME, and TIMESTAMP limitations

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	9999-12-31-23.59.59.999999

Limitations on identifier length

The following table lists limitations on identifier lengths in Derby.

Table 157. Identifier length limitations

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

The following table lists limitations on the numeric values in Derby.

Table 158. Numeric limitations

Value	Limit
Largest negative INTEGER	-2,147,483,648
Largest positive INTEGER	2,147,483,647

Value	Limit
Largest negative BIGINT	-9,223,372,036,854,775,808
Largest positive BIGINT	9,223,372,036,854,775,807
Largest negative SMALLINT	-32,768
Largest positive SMALLINT	32,767
Largest decimal precision	31
Largest negative DOUBLE	-1.7976931348623157E+308
Largest positive DOUBLE	1.7976931348623157E+308
Smallest negative normalized DOUBLE	-2.2250738585072014E-308
Smallest positive normalized DOUBLE	2.2250738585072014E-308
Smallest negative denormalized DOUBLE	-4.9E-324
Smallest positive denormalized DOUBLE	4.9E-324
Largest negative REAL	-3.4028235E+38
Largest positive REAL	3.4028235E+38
Smallest negative normalized REAL	-1.17549435E-38
Smallest positive normalized REAL	1.17549435E-38
Smallest negative denormalized REAL	-1.4E-45
Smallest positive denormalized REAL	1.4E-45

String limitations

The following table lists limitations on string values in Derby.

Table 159. String limitations

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

XML limitations

The following table shows the limitation on XML data types in Derby.

Table 160. XML limitations

Issue	Limitation
Length of XML	2,147,483,647 characters

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.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

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.