HyperSQL User Guide

HyperSQL Database Engine, aka HSQLDB

Edited by The HSQL Development Group, Blaine Simpson, and Fred Toussi

HyperSQL User Guide: HyperSQL Database Engine, aka HSQLDB

by The HSQL Development Group, Blaine Simpson, and Fred Toussi

$Revision: 3601 $

Published $Date: 2010-05-31 20:17:47 -0400 (Mon, 31 May 2010) $

HyperSQL

Table of Contents

List of Tables

List of Examples

Preface

HSQLDB (HyperSQL DataBase) is a modern relational database manager that conforms closely to the SQL:2008 Standard and JDBC 4 specifications. It supports all core features and many of the optional features of SQL:2008.

The first versions of HSQLDB were released in 2001. Version 2.0, first released in 2010, includes a complete rewrite of most parts of the database engine.

This documentation covers HyperSQL version 2.0. This documentation is regularly improved and undated. The latest, updated version can be found at http://hsqldb.org/doc/2.0/

If you notice any mistakes in this document, or if you have problems with the procedures themselves, please use the HSQLDB support facilities which are listed at http://hsqldb.org/support

Available formats for this document

This document is available in several formats.

You may be reading this document right now at http://hsqldb.org/doc/2.0, or in a distribution somewhere else. I hereby call the document distribution from which you are reading this, your *current distro*.

http://hsqldb.org/doc/2.0 hosts the latest production versions of all available formats. If you want a different format of the same *version* of the document you are reading now, then you should try your current distro. If you want the latest production version, you should try http://hsqldb.org/doc/2.0.

Sometimes, distributions other than http://hsqldb.org/doc/2.0 do not host all available formats. So, if you can't access the format that you want in your current distro, you have no choice but to use the newest production version at http://hsqldb.org/doc/2.0.

Table 1. Available formats of this document

format	your distro	at http://hsqldb.org/doc/2.0
Chunked HTML	index.html	http://hsqldb.org/doc/2.0/guide/
All-in-one HTML	guide.html	http://hsqldb.org/doc/2.0/guide/guide.html
PDF	guide.pdf	http://hsqldb.org/doc/2.0/guide/guide.pdf

If you are reading this document now with a standalone PDF reader, the your distro links may not work.

Chapter 1. Running and Using HyperSQL

Fred Toussi, The HSQL Development Group

$Revision: 3601 $

Copyright 2002-2010 Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2010-05-31 20:17:47 -0400 (Mon, 31 May 2010) $

The HSQLDB Jar

The HSQLDB jar package is located in the /lib directory of the ZIP package and contains several components and programs.

Components of the Hsqldb jar package

• HyperSQL RDBMS Engine (HSQLDB)

• HyperSQL JDBC Driver

• Database Manager (GUI database access tool, with Swing and AWT versions)

• Sql Tool (command line database access tool)

The HyperSQL RDBMS and JDBC Driver provide the core functionality. An additional jar contains Sql Tool (command line database access tool). SqlTool and the DatabaseManagers are general-purpose database tools that can be used with any database engine that has a JDBC driver.

Running Database Access Tools

The tools are used for interactive user access to databases, including creation of a database, inserting or modifying data, or querying the database. All tools are run in the normal way for Java programs. In the following example the Swing version of the Database Manager is executed. The `hsqldb.jar` is located in the directory `../lib` relative to the current directory.

```
java -cp ../lib/hsqldb.jar org.hsqldb.util.DatabaseManagerSwing
```

If `hsqldb.jar` is in the current directory, the command would change to:

```
java -cp hsqldb.jar org.hsqldb.util.DatabaseManagerSwing
```

Main classes for the Hsqldb tools

• `org.hsqldb.util.DatabaseManager`

• `org.hsqldb.util.DatabaseManagerSwing`

When a tool is up and running, you can connect to a database (may be a new database) and use SQL commands to access and modify the data.

Tools can use command line arguments. You can add the command line argument --help to get a list of available arguments for these tools.

A HyperSQL Database

Each HyperSQL database is called a catalog. There are three types of catalog depending on how the data is stored.

Types of catalog data

- *mem:* stored entirely in RAM - without any persistence beyond the JVM process's life

- *file:* stored in filesystem files

- *res:* stored in a Java resource, such as a Jar and always read-only

All-in-memory, *mem:* catalogs can be used for test data or as sophisticated caches for an application. These databases do not have any files.

A *file*: catalog consists of between 2 to 5 files, all named the same but with different extensions, located in the same directory. For example, the database named "test" consists of the following files:

- `test.properties`

- `test.script`

- `test.log`

- `test.data`

- `test.backup`

- `test.lobs`

The properties file contains a few settings about the database. The script file contains the definition of tables and other database objects, plus the data for non-cached tables. The log file contains recent changes to the database. The data file contains the data for cached tables and the backup file is a compressed backup of the last known consistent state of the data file. All these files are essential and should never be deleted. For some catalogs, the `test.data` and `test.backup` files will not be present. In addition to those files, a HyperSQL database may link to any formatted text files, such as CSV lists, anywhere on the disk.

While the "test" catalog is open, a `test.log` file is used to write the changes made to data. This file is removed at a normal SHUTDOWN. Otherwise (with abnormal shutdown) this file is used at the next startup to redo the changes. A `test.lck` file is also used to record the fact that the database is open. This is deleted at a normal SHUTDOWN.

> **Note**
>
> When the engine closes the database at a shutdown, it creates temporary files with the extension `.new` which it then renames to those listed above. In some circumstances, a `test.data.old` is created and deleted afterwards. These files should not be deleted by the user. At the time of the next startup, all such files will be deleted by the database engine.

A *res:* catalog consists of the files for a small, read-only database that can be stored inside a Java resource such as a ZIP or JAR archive and distributed as part of a Java application program.

In-Process Access to Database Catalogs

In general, JDBC is used for all access to databases. This is done by making a connection to the database, then using various methods of the `java.sql.Connection` object that is returned to access the data. Access to an *in-process* database is started from JDBC, with the database path specified in the connection URL. For example, if the *file:*

database name is "testdb" and its files are located in the same directory as where the command to run your application was issued, the following code is used for the connection:

```
Connection c = DriverManager.getConnection("jdbc:hsqldb:file:testdb", "SA", "");
```

The database file path format can be specified using forward slashes in Windows hosts as well as Linux hosts. So relative paths or paths that refer to the same directory on the same drive can be identical. For example if your database path in Linux is `/opt/db/testdb` and you create an identical directory structure on the `C:` drive of a Windows host, you can use the same URL in both Windows and Linux:

```
Connection c = DriverManager.getConnection("jdbc:hsqldb:file:/opt/db/testdb", "SA", "");
```

When using relative paths, these paths will be taken relative to the directory in which the shell command to start the Java Virtual Machine was executed. Refer to the Javadoc for `JDBCConnection` for more details.

Paths and database names for file databases are treated as case-sensitive when the database is created or the first connection is made to the database. But if a second connection is made to an open database, using a path and name that differs only in case, then the connection is made to the existing open database. This measure is necessary because in Windows the two paths are equivalent.

A *mem:* database is specified by the *mem:* protocol. For *mem:* databases, the path is simply a name. Several *mem:* databases can exist at the same time and distinguished by their names. In the example below, the database is called "mymemdb":

```
Connection c = DriverManager.getConnection("jdbc:hsqldb:mem:mymemdb", "SA", "");
```

A *res:* database, is specified by the *res:* protocol. As it is a Java resource, the database path is a Java URL (similar to the path to a class). In the example below, "resdb" is the root name of the database files, which exists in the directory "org/my/path" within the classpath (probably in a Jar). A Java resource is stored in a compressed format and is decompressed in memory when it is used. For this reason, a *res:* database should not contain large amounts of data and is always read-only.

```
Connection c = DriverManager.getConnection("jdbc:hsqldb:res:org.my.path.resdb", "SA", "");
```

The first time *in-process* connection is made to a database, some general data structures are initialised and a few helper threads are started. After this, creation of connections and calls to JDBC methods of the connections execute as if they are part of the Java application that is making the calls. When the SQL command "SHUTDOWN" is executed, the global structures and helper threads for the database are destroyed.

Note that only one Java process at a time can make *in-process* connections to a given *file:* database. However, if the *file:* database has been made read-only, or if connections are made to a *res:* database, then it is possible to make *in-process* connections from multiple Java processes.

Listener / Server Modes

For most applications, *in-process* access is faster, as the data is not converted and sent over the network. The main drawback is that it is not possible by default to connect to the database from outside your application. As a result you cannot check the contents of the database with external tools such as Database Manager while your application is running.

Server modes provide the maximum accessibility. The database engine runs in a JVM and opens one or more *in-process* catalogs. It listens for connections from programs on the same computer or other computers on the network. It translates these connections into *in-process* connections to the databases.

Several different programs can connect to the server and retrieve or update information. Applications programs (clients) connect to the server using the HyperSQL JDBC driver. In most server modes, the server can serve an unlimited number of databases that are specified at the time of running the server, or optionally, as a connection request is received.

A Sever mode is also the prefered mode of running the database during development. It allows you to query the database from a separate database access utility while your application is running.

There are three server modes, based on the protocol used for communications between the client and server. They are briefly discussed below. More details on servers is provided in the HyperSQL Network Listeners chapter.

HyperSQL HSQL Server

This is the preferred way of running a database server and the fastest one. A proprietary communications protocol is used for this mode. A command similar to those used for running tools and described above is used for running the server. The following example of the command for starting the server starts the server with one (default) database with files named "mydb.*" and the public name of "xdb". The public name hides the file names from users.

```
java -cp ../lib/hsqldb.jar org.hsqldb.server.Server --database.0 file:mydb --dbname.0 xdb
```

The command line argument `--help` can be used to get a list of available arguments.

HyperSQL HTTP Server

This method of access is used when the computer hosting the database server is restricted to the HTTP protocol. The only reason for using this method of access is restrictions imposed by firewalls on the client or server machines and it should not be used where there are no such restrictions. The HyperSQL HTTP Server is a special web server that allows JDBC clients to connect via HTTP. The server can also act as a small general-purpose web server for static pages.

To run an HTTP server, replace the main class for the server in the example command line above with the following:

```
org.hsqldb.server.WebServer
```

The command line argument `--help` can be used to get a list of available arguments.

HyperSQL HTTP Servlet

This method of access also uses the HTTP protocol. It is used when a separate servlet engine (or application server) such as Tomcat or Resin provides access to the database. The Servlet Mode cannot be started independently from the servlet engine. The `Servlet` class, in the HSQLDB jar, should be installed on the application server to provide the connection. The database is specified using an application server property. Refer to the source file   `src/org/hsqldb/server/Servlet.java` to see the details.

Both HTTP Server and Servlet modes can only be accessed using the JDBC driver at the client end. They do not provide a web front end to the database. The Servlet mode can serve only a single database.

Please note that you do not normally use this mode if you are using the database engine in an application server. In this situation, connections to a catalog are usually made *in-process*, or using a separate Server

Connecting to a Database Server

When a HyperSQL server is running, client programs can connect to it using the HSQLDB JDBC Driver contained in `hsqldb.jar`. Full information on how to connect to a server is provided in the Java Documentation for   `JD-BCConnection` (located in the `/doc/apidocs` directory of HSQLDB distribution). A common example is connection to the default port (9001) used for the *hsql:* protocol on the same machine:

Example 1.1. Java code to connect to the local hsql Server

```
try {
    Class.forName("org.hsqldb.jdbc.JDBCDriver" );
```

```
} catch (Exception e) {
    System.err.println("ERROR: failed to load HSQLDB JDBC driver.");
    e.printStackTrace();
    return;
}

Connection c = DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/xdb", "SA", "");
```

If the HyperSQL HTTP server is used, the protocol is *http:* and the URL will be different:

Example 1.2. Java code to connect to the local http Server

```
Connection c = DriverManager.getConnection("jdbc:hsqldb:http://localhost/xdb", "SA", "");
```

Note in the above connection URL, there is no mention of the database file, as this was specified when running the server. Instead, the public name defined for dbname.0 is used. Also, see the HyperSQL Network Listeners chapter for the connection URL when there is more than one database per server instance.

Security Considerations

When a HyperSQL server is run, network access should be adequately protected. Source IP addresses may be restricted by use of our Access Control List feature , network filtering software, firewall software, or standalone firewalls. Only secure passwords should be used-- most importantly, the password for the default system user should be changed from the default empty string. If you are purposefully providing data to the public, then the wide-open public network connection should be used exclusively to access the public data via read-only accounts. (i.e., neither secure data nor privileged accounts should use this connection). These considerations also apply to HyperSQL servers run with the HTTP protocol.

HyperSQL provides two optional security mechanisms. The encrypted SSL protocol , and Access Control Lists . Both mechanisms can be specified when running the Server or WebServer. From the client, the URL's co connect to an SSL server is slightly different:

Example 1.3. Java code to connect to the local secure SSL hsql and http Servers

```
Connection c = DriverManager.getConnection("jdbc:hsqldb:hsqls://localhost/xdb", "SA", "");
Connection c = DriverManager.getConnection("jdbc:hsqldb:https://localhost/xdb", "SA", "");
```

The security features are discussed in detail in the listeners chapter.

Using Multiple Databases

A server can provide connections to more than one database. In the examples above, more than one set of database names can be specified on the command line. It is also possible to specify all the databases in a `.properties` file, instead of the command line. These capabilities are covered in the HyperSQL Network Listeners chapter

Accessing the Data

As shown so far, a `java.sql.Connection` object is always used to access the database. But the speed and performance depends on the type of connection.

Establishing a connection and closing it has some overheads, therefore it is not good practice to create a new connection to perform a small number of operations. A connection should be reused as much as possible and closed only when it is not going to be used again for a long while.

Reuse is more important for server connections. A server connection uses a TCP port for communications. Each time a connection is made, a port is allocated by the operating system and deallocated after the connection is closed. If

many connections are made from a single client, the operating system may not be able to keep up and may refuse the connection attempt.

A `java.sql.Connection` object has some methods that return further `java.sql.*` objects. All these objects belong to the connection that returned them and are closed when the connection is closed. These objects can be reused, but if they are not needed after performing the operations, they should be closed.

A `java.sql.DatabaseMetaData` object is used to get metadata for the database.

A `java.sql.Statement` object is used to execute queries and data change statements. A `java.sql.Statement` can be reused to execute a different statement each time.

A `java.sql.PreparedStatement` object is used to execute a single statement repeatedly. The SQL statement usually contains parameters, which can be set to new values before each reuse. When a `java.sql.PreparedStatement` object is created, the engine keeps the compiled SQL statement for reuse, until the `java.sql.PreparedStatement` object is closed. As a result, repeated use of a `java.sql.PreparedStatement` is much faster than using a `java.sql.Statement` object.

A `java.sql.CallableStatement` object is used to execute an SQL CALL statement. The SQL CALL statement may contain parameters, which should be set to new values before each reuse. Similar to `java.sql.PreparedStatement`, the engine keeps the compiled SQL statement for reuse, until the `java.sql.CallableStatement` object is closed.

A `java.sql.Connection` object also has some methods for transaction control.

The `commit()` method performs a COMMIT while the `rollback()` method performs a ROLLBACK SQL statement.

The `setSavepoint(String name)` method performs a SAVEPOINT `<name>` SQL statement and returns a `java.sql.Savepoint` object. The `rollback(Savepoint name)` method performs a ROLLBACK TO SAVEPOINT `<name>` SQL statement.

The Javadoc for `JDBCConnection`, `JDBCDriver`, `JDBCDatabaseMetadata` `JDBCResultSet`, `JDBCStatement`, `JDBCPreparedStatement` list all the supported JDBC methods together with information that is specific to HSQLDB.

Closing the Database

All databases running in different modes can be closed with the SHUTDOWN command, issued as an SQL statement.

When SHUTDOWN is issued, all active transactions are rolled back. The catalog files are then saved in a form that can be opened quickly the next time the catalog is opened.

A special form of closing the database is via the SHUTDOWN COMPACT command. This command rewrites the `.data` file that contains the information stored in CACHED tables and compacts it to its minimum size. This command should be issued periodically, especially when lots of inserts, updates or deletes have been performed on the cached tables. Changes to the structure of the database, such as dropping or modifying populated CACHED tables or indexes also create large amounts of unused file space that can be reclaimed using this command.

Databases are not closed when the last connection to the database is explicitly closed via JDBC. A connection property, `shutdown=true`, can be specified on the first connection to the database (the connection that opens the database) to force a shutdown when the last connection closes.

Example 1.4. specifying a connection property to shutdown the database when the last connection is closed

```
Connection c = DriverManager.getConnection(
```

```
          "jdbc:hsqldb:file:/opt/db/testdb;shutdown=true", "SA", "");
```

This feature is useful for running tests, where it may not be practical to shutdown the database after each test. But it is not recommended for application programs.

Creating a New Database

When a server instance is started, or when a connection is made to an *in-process* database, a new, empty database is created if no database exists at the given path.

With HyperSQL 2.0 the username and password that are specified for the connection are used for the new database. Both the username and password are case-sensitive. (The exception is the default SA user, which is not case-sensitive). If no username or password is specified, the default SA user and an empty password are used.

This feature has a side effect that can confuse new users. If a mistake is made in specifying the path for connecting to an existing database, a connection is nevertheless established to a new database. For troubleshooting purposes, you can specify a connection property ifexists=true to allow connection to an existing database only and avoid creating a new database. In this case, if the database does not exist, the getConnection() method will throw an exception.

Example 1.5. specifying a connection property to disallow creating a new database

```
  Connection c = DriverManager.getConnection(
        "jdbc:hsqldb:file:/opt/db/testdb;ifexists=true", "SA", "");
```

A database has many optional properties, described in the System Management and Deployment Issues chapter. You can specify most of these properties on the URL or in the connection properties for the first connection that creates the database. See the Properties chapter.

Chapter 2. SQL Language

Fred Toussi, The HSQL Development Group

$Revision: 3601 $

Copyright 2002-2010 Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2010-05-31 20:17:47 -0400 (Mon, 31 May 2010) $

Standards Support

HyperSQL 2.0 supports the dialect of SQL defined by SQL standards 92, 1999, 2003 and 2008. This means where a feature of the standard is supported, e.g. left outer join, the syntax is that specified by the standard text. Almost all syntactic features of SQL-92 up to Advanced Level are supported, as well as SQL:2008 core and many optional features of this standard. Work is in progress for a formal declaration of conformance.

At the time of this release, HyperSQL supports the widest range of SQL standard features among all open source RDBMS.

Various chapters of this guide list the supported syntax. When writing or converting existing SQL DDL (Data Definition Language), DML (Data Manipulation Language) or DQL (Data Query Language) statements for HSQLDB, you should consult the supported syntax and modify the statements accordingly. Some statements written for older versions may have to be modified.

Over 300 words are reserved by the standard and should not be used as table or column names. For example, the word POSITION is reserved as it is a function defined by the Standards with a similar role as `String.indexOf()` in Java. HyperSQL does not currently prevent you from using a reserved word if it does not support its use or can distinguish it. For example CUBE is a reserved words that is not currently supported by HyperSQL and is allowed as a table or column name. You should avoid using such names as future versions of HyperSQL are likely to support the reserved words and may reject your table definitions or queries. The full list of SQL reserved words is in the appendix Lists of Keywords .

If you have to use a reserved keyword as the name of a database object, you can enclose it in double quotes.

HyperSQL also supports enhancements with keywords and expressions that are not part of the SQL standard. Expressions such as `SELECT TOP 5 FROM ..`, `SELECT LIMIT 0 10 FROM ...` or `DROP TABLE mytable IF EXISTS` are among such constructs.

Many print books cover SQL Standard syntax and can be consulted. For a well-written basic guide to SQL with examples, you can also consult PostgreSQL: Introduction and Concepts [http://www.postgresql.org/files/documentation/books/aw_pgsql/index.html] by Bruce Momjian, which is available on the web. Most of the core SQL coverage in the book applies also to HyperSQL. There are some differences in keywords supported by one and not the other engine (OUTER, OID's, etc.) or used differently (IDENTITY/SERIAL, TRIGGER, SEQUENCE, etc.).

In HyperSQL version 2.0, all features of JDBC4 that apply to the capabilities of HSQLDB are fully supported. The relevant JDBC classes are thoroughly documented with additional clarifications and HyperSQL specific comments. See the JavaDoc for the `org.hsqldb.jdbc.*` classes.

SQL Data and Tables

In an SQL system, all significant data is stored in tables and sequence generators. Therefore, the first step in creating a database is defining the tables and their columns. The SQL standard supports temporary tables, which are for temporary data, and permanent base tables, which are for persistent data.

Temporary Tables

TEMPORARY tables are not saved and last only for the lifetime of the Connection object. The contents of each TEMP table is visible only from the Connection that was used to populate it. The definition of TEMP tables conforms to the GLOBAL TEMPORARY type in the SQL standard. The definition of the table persists but each new connections sees its own copy of the table, which is empty at the beginning. When the connection commits, the contents of the table are cleared by default. If the table definition statements includes ON COMMIT PRESERVE ROWS, then the contents are kept when a commit takes place.

Persistent Tables

HSQLDB supports the Standard definition of persistent base table, but defines three types according to the way the data is stored. These are MEMORY tables, CACHED tables and TEXT tables.

Memory tables are the default type when the CREATE TABLE command is used. Their data is held entirely in memory but any change to their structure or contents is written to the `*.log` and `*.script` files. The `*.script` file and the `*.log` file are read the next time the database is opened, and the MEMORY tables are recreated with all their contents. So unlike TEMPORARY tables, MEMORY tables are persistent. When the database is opened, all the data for the memory tables is read and inserted. This process may take a long time if the database is larger than tens of megabytes. When the database is shutdown, all the data is saved. This can also take a long time.

CACHED tables are created with the CREATE CACHED TABLE command. Only part of their data or indexes is held in memory, allowing large tables that would otherwise take up to several hundred megabytes of memory. Another advantage of cached tables is that the database engine takes less time to start up when a cached table is used for large amounts of data. The disadvantage of cached tables is a reduction in speed. Do not use cached tables if your data set is relatively small. In an application with some small tables and some large ones, it is better to use the default, MEMORY mode for the small tables.

TEXT tables use a CSV (Comma Separated Value) or other delimited text file as the source of their data. You can specify an existing CSV file, such as a dump from another database or program, as the source of a TEXT table. Alternatively, you can specify an empty file to be filled with data by the database engine. TEXT tables are efficient in memory usage as they cache only part of the text data and all of the indexes. The Text table data source can always be reassigned to a different file if necessary. The commands are needed to set up a TEXT table as detailed in the Text Tables chapter.

With all-in-memory databases, both MEMORY table and CACHED table declarations are treated as declarations for non-persistent memory tables. TEXT table declarations are not allowed in these databases.

The default type of table resulting from future CREATE TABLE statements can be specified with the SQL command:

```
SET DATABASE DEFAULT TABLE TYPE { CACHED | MEMORY };
```

The type of an existing table can be changed with the SQL command:

```
SET TABLE <table name> TYPE { CACHED | MEMORY };
```

SQL statements access different types of tables uniformly. No change to statements is needed to access different types of table.

Lob Data

Lobs are logically stored in columns of tables. Their physical storage is a separate *.lobs file. In version 2.0 this file is created as soon as a BLOB or CLOB is inserted into the database. The file will grow as new lobs are inserted into the database. In version 2.0, the *.lobs file is never deleted even if all lobs are deleted from the database (In this case you can delete the .lobs file after a SHTUDOWN).

Basic Types and Operations

HyperSQL supports all the types defined by SQL-92, plus BOOLEAN, BINARY and LOB types that were added later to the SQL Standard. It also supports the non-standard OTHER type to store serializable Java objects.

SQL is a strongly typed language. All data stored in specific columns of tables and other objects (such as sequence generators) have specific types. Each data item conforms to the type limits such as precision and scale for the column. It also conforms to any additional integrity constraints that are defined as CHECK constraints in domains or tables. Types can be explicitly converted using the CAST expression, but in most expressions they are converted automatically.

Data is returned to the user (or the application program) as a result of executing SQL statements such as query expressions or function calls. All statements are compiled prior to execution and the return type of the data is known after compilation and before execution. Therefore, once a statement is prepared, the data type of each column of the returned result is known, including any precision or scale property. The type does not change when the same query that returned one row, returns many rows as a result of adding more data to the tables.

Some SQL functions used within SQL statements are polymorphic, but the exact type of the argument and the return value is determined at compile time.

When a statement is prepared, using a JDBC PreparedStatement object, it is compiled by the engine and the type of the columns of its ResultSet and / or its parameters are accessible through the methods of PreparedStatement.

Numeric Types

TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC and DECIMAL (without a decimal point) are the supported integral types. They correspond respectively to `byte`, `short`, `int`, `long`, `BigDecimal` and `BigDecimal` Java types in the range of values that they can represent (NUMERIC and DECIMAL are equivalent). The type TINYINT is an HSQLDB extension to the SQL Standard, while the others conform to the Standard definition. The SQL type dictates the maximum and minimum values that can be held in a field of each type. For example the value range for TINYINT is -128 to +127. The bit precision of TINYINT, SMALLINT, INTEGER and BIGINT is respectively 8, 16, 32 and 64. For NUMERIC and DECIMAL, decimal precision is used.

DECIMAL and NUMERIC with decimal fractions are mapped to `java.math.BigDecimal` and can have very large numbers of digits. In HyperSQL the two types are equivalent. These types, together with integral types, are called exact numeric types.

In HyperSQL, REAL, FLOAT, DOUBLE are equivalent and all mapped to `double` in Java. These types are defined by the SQL Standard as approximate numeric types. The bit-precision of all these types is 64 bits.

The decimal precision and scale of NUMERIC and DECIMAL types can be optionally defined. For example, DECIMAL(10,2) means maximum total number of digits is 10 and there are always 2 digits after the decimal point, while DECIMAL(10) means 10 digits without a decimal point. The bit-precision of FLOAT can also be defined, but in this case, it is ignored and the default bit-precision of 64 is used. The default precision of NUMERIC and DECIMAL (when not defined) is 100.

Note: If a database has been set to ignore type precision limits with the SET DATABASE SQL SIZE FALSE command, then a type definition of DECIMAL with no precision and scale is treated as DECIMAL(100,10). In normal operation, it is treated as DECIMAL(100).

Integral Types

In expressions, TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC and DECIMAL (without a decimal point) are fully interchangeable, and no data narrowing takes place.

If the SELECT statement refers to a simple column or function, then the return type is the type corresponding to the column or the return type of the function. For example:

```
    CREATE TABLE t(a INTEGER, b BIGINT);
    SELECT MAX(a), MAX(b) FROM t;
```

will return a `ResultSet` where the type of the first column is `java.lang.Integer` and the second column is `java.lang.Long`. However,

```
    SELECT MAX(a) + 1, MAX(b) + 1 FROM t;
```

will return `java.lang.Long` and `BigDecimal` values, generated as a result of uniform type promotion for all the return values. Note that type promotion to `BigDecimal` ensures the correct value is returned if `MAX(b)` evaluates to `Long.MAX_VALUE`.

There is no built-in limit on the size of intermediate integral values in expressions. As a result, you should check for the type of the `ResultSet` column and choose an appropriate `getXXXX()` method to retrieve it. Alternatively, you can use the `getObject()` method, then cast the result to `java.lang.Number` and use the `intValue()` or `longValue()` methods on the result.

When the result of an expression is stored in a column of a database table, it has to fit in the target column, otherwise an error is returned. For example when `1234567890123456789012 / 12345687901234567890` is evaluated, the result can be stored in any integral type column, even a TINYINT column, as it is a small value.

In SQL Statements, an integer literal is treated as INTEGER, unless its value does not fit. In this case it is treated as BIGINT or DECIMAL, depending on the value.

Depending on the types of the operands, the result of the operations is returned in a JDBC `ResultSet` in any of related Java types: `Integer`, `Long` or `BigDecimal`. The `ResultSet.getXXXX()` methods can be used to retrieve the values so long as the returned value can be represented by the resulting type. This type is deterministically based on the query, not on the actual rows returned.

Other Numeric Types

In SQL statements, number literals with a decimal point are treated as DECIMAL unless they are written with an exponent. Thus `0.2` is considered a DECIMAL value but `0.2E0` is considered a DOUBLE value.

When an approximate numeric type, REAL, FLOAT or DOUBLE (all synonymous) is part of an expression involving different numeric types, the type of the result is DOUBLE. DECIMAL values can be converted to DOUBLE unless they are beyond the `Double.MIN_VALUE - Double.MAX_VALUE` range. For example, A * B, A / B, A + B, etc. will return a DOUBLE value if either A or B is a DOUBLE.

Otherwise, when no DOUBLE value exists, if a DECIMAL or NUMERIC value is part an expression, the type of the result is DECIMAL or NUMERIC. Similar to integral values, when the result of an expression is assigned to a table column, the value has to fit in the target column, otherwise an error is returned. This means a small, 4 digit value of DECIMAL type can be assigned to a column of SMALLINT or INTEGER, but a value with 15 digits cannot.

When a DECIMAL values is multiplied by a DECIMAL or integral type, the resulting scale is the sum of the scales of the two terms. When they are divided, the result is a value with a scale (number of digits to the right of the decimal point) equal to the larger of the scales of the two terms. The precision for both operations is calculated (usually increased) to allow all possible results.

The distinction between DOUBLE and DECIMAL is important when a division takes place. For example, `10.0/8.0` (DECIMAL) equals `1.2` but `10.0E0/8.0E0` (DOUBLE) equals `1.25`. Without division operations, DECIMAL values represent exact arithmetic.

REAL, FLOAT and DOUBLE values are all stored in the database as `java.lang.Double` objects. Special values such as NaN and +-Infinity are also stored and supported. These values can be submitted to the database via JDBC `PreparedStatement` methods and are returned in `ResultSet` objects. The result can be retrieved from a Re-

sultSet in the required type so long as it can be represented. When `PreparedStatement.setDouble()` or `setFloat()` is used, the value is treated as a DOUBLE automatically.

In short,

```
<numeric type> ::= <exact numeric type> | <approximate numeric type>

<exact numeric type> ::= NUMERIC [ <left paren> <precision> [ <comma> <scale> ]
<right paren> ] | { DECIMAL | DEC } [ <left paren> <precision> [ <comma> <scale> ]
<right paren> ] | SMALLINT | INTEGER | INT | BIGINT

<approximate numeric type> ::= FLOAT [ <left paren> <precision> <right paren> ]
| REAL | DOUBLE PRECISION

<precision> ::= <unsigned integer>

<scale> ::= <unsigned integer>
```

Boolean Type

The BOOLEAN type conforms to the SQL Standard and represents the values `TRUE`, `FALSE` and `UNKNOWN`. This type of column can be initialised with Java boolean values, or with `NULL` for the `UNKNOWN` value.

The three-value logic is sometimes misunderstood. For example, x IN (1, 2, NULL) does not return true if x is NULL.

In previous versions of HyperSQL, BIT was simply an alias for BOOLEAN. In version 2.0, BIT is a single-bit bit map.

```
<boolean type> ::= BOOLEAN
```

The SQL Standard does not support type conversion to BOOLEAN apart from character strings that consists of boolean literals. Because the BOOLEAN type is relatively new to the Standard, several database products used other types to represent boolean values. For improved compatibility, HyperSQL allows some type conversions to boolean.

Values of BIT and BIT VARYING types with length 1 can be converted to BOOLEAN. If the bit is set, the result of conversion is the TRUE value, otherwise it is FALSE.

Values of TINYINT, SMALLINT, INTEGER and BIGINT types can be converted to BOOLEAN. If the value is zero, the result is the FALSE value, otherwise it is TRUE.

Character String Types

The CHARACTER, CHARACTER VARYING and CLOB types are the SQL Standard character string types. CHAR, VARCHAR and CHARACTER LARGE OBJECT are synonyms for these types. HyperSQL also supports LONG-VARCHAR as a synonym for VARCHAR. If LONGVARCHAR is used without a length, then a length of 1M is assigned.

HyperSQL's default character set is Unicode, therefore all possible character strings can be represented by these types.

The SQL Standard behaviour of the CHARACTER type is a remnant of legacy systems in which character strings are padded with spaces to fill a fixed width. These spaces are sometimes significant while in other cases they are silently discarded. It would be best to avoid the CHARACTER type altogether. With the rest of the types, the strings are not padded when assigned to columns or variables of the given type. The trailing spaces are still considered discardable for all character types. Therefore if a string with trailing spaces is too long to assign to a column or variable of a given length, the spaces beyond the type length are discarded and the assignment succeeds (provided all the characters beyond the type length are spaces).

The VARCHAR and CLOB types have length limits, but the strings are not padded by the system. Note that if you use a large length for a VARCHAR or CLOB type, no extra space is used in the database. The space used for each stored item is proportional to its actual length.

If CHARACTER is used without specifying the length, the length defaults to 1. For the CLOB type, the length limit can be defined in units of kilobyte (K, 1024), megabyte (M, 1024 * 1024) or gigabyte (G, 1024 * 1024 * 1024), using the `<multiplier>`. If CLOB is used without specifying the length, the length defaults to 1M.

```
<character string type> ::= { CHARACTER | CHAR } [ <left paren> <character
length> <right paren> ] | { CHARACTER VARYING | CHAR VARYING | VARCHAR } <left
paren> <character length> <right paren> | LONGVARCHAR [ <left paren> <character
length> <right paren> ] | <character large object type>

<character large object type> ::= { CHARACTER LARGE OBJECT | CHAR LARGE OBJECT
| CLOB } [ <left paren> <character large object length> <right paren> ]

<character length> ::= <unsigned integer> [ <char length units> ]

<large object length> ::= <length> [ <multiplier> ] | <large object length token>

<character large object length> ::= <large object length> [ <char length units> ]

<large object length token> ::= <digit>... <multiplier>

<multiplier> ::= K | M | G

<char length units> ::= CHARACTERS | OCTETS
```

```
CHAR(10)
CHARACTER(10)
VARCHAR(2)
CHAR VARYING(2)
CLOB(1000)
CLOB(30K)
CHARACTER LARGE OBJECT(1M)
LONGVARCHAR
```

Binary String Types

The BINARY, BINARY VARYING and BLOB types are the SQL Standard binary string types. VARBINARY and BINARY LARGE OBJECT are synonyms for BINARY VARYING and BLOB types. HyperSQL also supports LONGVARBINARY as a synonym for VARBINARY.

Binary string types are used in a similar way to character string types. There are several built-in functions that are overloaded to support character, binary and bit strings.

The BINARY type represents a fixed width-string. Each shorter string is padded with zeros to fill the fixed width. Similar to the CHARACTER type, the trailing zeros in the BINARY string are simply discarded in some operations. For the same reason, it is best to avoid this particular type.

If BINARY is used without specifying the length, the length defaults to 1. For the BLOB type, the length limit can be defined in units of kilobyte (K, 1024), megabyte (M, 1024 * 1024) or gigabyte (G, 1024 * 1024 * 1024), using the `<multiplier>`. If BLOB is used without specifying the length, the length defaults to 1M.

```
<binary string type> ::= BINARY [ <left paren> <length> <right paren> ] | { BINARY
VARYING | VARBINARY } <left paren> <length> <right paren> | LONGVARBINARY [ <left
paren> <length> <right paren> ] | <binary large object string type>
```

```
<binary large object string type> ::= { BINARY LARGE OBJECT | BLOB } [ <left
paren> <large object length> <right paren> ]
```

```
<length> ::= <unsigned integer>
```

```
BINARY(10)
VARBINARY(2)
BINARY VARYING(2)
BLOB(1000)
BLOB(30K)
BINARY LARGE OBJECT(1M)
LONGVARBINARY
```

Bit String Types

The BIT and BIT VARYING types are the supported bit string types. These types were defined by SQL:1999 but were later removed from the Standard. Bit types represent bit maps of given lengths. Each bit is 0 or 1. The BIT type represents a fixed width-string. Each shorter string is padded with zeros to fill the fixed with. If BIT is used without specifying the length, the length defaults to 1. The BIT VARYING type has a maximum width and shorter strings are not padded.

Before the introduction of the BOOLEAN type to the SQL Standard, a sigle-bit string of the type BIT(1) was commonly used. For compatibility with other products that do not conform to, or extend, the SQL Standard, HyperSQL allows values of BIT and BIT VARYING types with length 1 to be converted to and from the BOOLEAN type. BOOLEAN TRUE is considered equal to B'1', BOOLEAN FALSE is considered equal to B'0'.

For the same reason, numeric values can be assigned to columns and variables of the type BIT(1). For assignment, the numeric value zero is converted to B'0', while all other values are converted to B'1'. For comparison, numeric values 1 is considered equal to B'1' and numeric value zero is considered equal to B'0'.

It is not allowed to perform other arithmetic or boolean operations involving BIT(1) and BIT VARYING(1). The kid of operations allowed on bit strings are analogous to those allowed on BINARY and CHARACTER strings. Several built-in functions support all three types of string.

```
<bit string type> ::= BIT [ <left paren> <length> <right paren> ] | BIT VARYING
<left paren> <length> <right paren>
```

```
BIT
BIT(10)
BIT VARYING(2)
```

Storage and Handling of Java Objects

Any serializable JAVA Object can be inserted directly into a column of type OTHER using any variation of `PreparedStatement.setObject()` methods.

For comparison purposes and in indexes, any two Java Objects are considered equal unless one of them is NULL. You cannot search for a specific object or perform a join on a column of type OTHER.

Please note that HSQLDB is not an object-relational database. Java Objects can simply be stored internally and no operations should be performed on them other than assignment between columns of type OTHER or tests for NULL. Tests such as `WHERE object1 = object2` do not mean what you might expect, as any non-null object would satisfy such a tests. But `WHERE object1 IS NOT NULL` is perfectly acceptable.

The engine does not allow normal column values to be assigned to Java Object columns (for example, assigning an INTEGER or STRING to such a column with an SQL statement such as `UPDATE mytable SET objectcol = intcol WHERE ...`).

```
<java object type> ::= OTHER
```

Type Length, Precision and Scale

In older version of HyperSQL, all table column type definitions with a column length, precision or scale qualifier were accepted and ignored. HSQLDB 1.8 enforced correctness but included an option to enforce the length, precision or scale.

In HyperSQL 2.0, length, precision and scale qualifiers are always enforced. For backward compatibility, when older databases which had the property hsqldb.enforce_strict_size=false are converted to version 2.0, this property is retained. However, this is a temporary measure. You should test your application to ensure the length, precision and scale that is used for column definitions is appropriate for the application data. You can test with the default database setting, which enforces the sizes.

String types, including all BIT, BINARY and CHAR string types plus CLOB and BLOB, are generally defined with a length. If no length is specified for BIT, BINARY and CHAR, the default length is 1. For CLOB and BLOB an implementation defined length of 1M is used.

TIME and TIMESTAMP types can be defined with a fractional second precision between 0 and 9. INTERVAL type definition may have precision and, in some cases, fraction second precision. DECIMAL and NUMERIC types may be defined with precision and scale. For all of these types a default precision or scale value is used if one is not specified. The default scale is 0. The default fractional precision for TIME is 0, while it is 6 for TIMESTAMP.

Values can be converted from one type to another in two different ways: by using explicit CAST expression or by implicit conversion used in assignment, comparison and aggregation.

String values cannot be assigned to VARCHAR columns if they are longer than the defined type length. For CHARACTER columns, a long string can be assigned (with truncation) only if all the characters after the length are spaces. Shorter strings are padded with the space character when inserted into a CHARACTER column. Similar rules are applied to VARBINARY and BINARY columns. For BINARY columns, the padding and truncation rules are applied with zero bytes, instead of spaces.

Explicit CAST of a value to a CHARACTER or VARCHAR type will result in forced truncation or padding. So a test such as `CAST (mycol AS VARCHAR(2)) = 'xy'` will find the values beginning with 'xy'. This is the equivalent of `SUBSTRING(mycol FROM 1 FOR 2)= 'xy'`.

For all numeric types, the rules of explicit cast and implicit conversion are the same. If cast or conversion causes any digits to be lost from the fractional part, it can take place. If the non-fractional part of the value cannot be represented in the new type, cast or conversion cannot take place and will result in a data exception.

There are special rules for DATE, TIME, TIMESTAMP and INTERVAL casts and conversions.

Datetime types

HSQLDB fully supports datetime and interval types and operations, including all relevant optional features, as specified by the SQL Standard since SQL-92. The two groups of types are complementary.

The DATE type represents a calendar date with YEAR, MONTH and DAY fields.

The TIME type represents time of day with HOUR, MINUTE and SECOND fields, plus an optional SECOND FRACTION field.

The TIMESTAMP type represents the combination of DATE and TIME types.

TIME and TIMESTAMP types can include WITH TIME ZONE or WITHOUT TIME ZONE (the default) qualifiers. They can have fractional second parts. For example, TIME(6) has six fractional digits for the second field.

If fractional second precision is not specified, it defaults to 0 for TIME and to 6 for TIMESTAMP.

```
<datetime type> ::= DATE | TIME [ <left paren> <time precision> <right paren> ]
[ <with or without time zone> ] | TIMESTAMP [ <left paren> <timestamp precision>
<right paren> ] [ <with or without time zone> ]

<with or without time zone> ::= WITH TIME ZONE | WITHOUT TIME ZONE

<time precision> ::= <time fractional seconds precision>

<timestamp precision> ::= <time fractional seconds precision>

<time fractional seconds precision> ::= <unsigned integer>
```

```
DATE
TIME(6)
TIMESTAMP(2) WITH TIME ZONE
```

Examples of the string literals used to represent date time values, some with time zone, some without, are below:

```
DATE '2008-08-22'
TIMESTAMP '2008-08-08 20:08:08'
TIMESTAMP '2008-08-08 20:08:08+8:00' /* Beijing */
TIME '20:08:08.034900'
TIME '20:08:08.034900-8:00' /* US Pacific */
```

Time Zone

DATE values do not take time zones. For example United Nations designates 5 June as World Environment Day, which was observed on DATE '2008-06-05' in different time zones.

TIME and TIMESTAMP values without time zone, usually have a context that indicates some local time zone. For example, a database for college course timetables usually stores class dates and times without time zones. This works because the location of the college is fixed and the time zone displacement is the same for all the values. Even when the events take place in different time zones, for example international flight times, it is possible to store all the datetime information as references to a single time zone, usually GMT. For some databases it may be useful to store the time zone displacement together with each datetime value. SQL's TIME WITH TIME ZONE and TIMESTAMP WITH TIME ZONE values include a time zone displacement value.

The time zone displacement is of the type INTERVAL HOUR TO MINUTE. This data type is described in the next section. The legal values are between '–14:00' and '+14:00'.

Operations on Datetime Types

The expression `<datetime expression> AT TIME ZONE <time displacement>` evaluates to a datetime value representing exactly the same point of time in the specified `<time displacement>`. The expression, `AT LOCAL` is equivalent to `AT TIME ZONE <local time displacement>`. If `AT TIME ZONE` is used with a datetime operand of type WITHOUT TIME ZONE, the operand is first converted to a value of type WITH TIME ZONE at the session's time displacement, then the specified time zone displacement is set for the value. Therefore, in these cases, the final value depends on the time zone of the session in which the statement was used.

AT TIME ZONE, modifies the field values of the datetime operand. This is done by the following procedure:

1. determine the corresponding datetime at UTC.

2. find the datetime value at the given time zone that corresponds with the UTC value from step 1.

Example a:

```
TIME '12:00:00' AT TIME ZONE INTERVAL '1:00' HOUR TO MINUTE
```

If the session's time zone displacement is -'8:00', then in step 1, TIME '12:00:00' is converted to UTC, which is TIME '20:00:00+0:00'. In step 2, this value is expressed as TIME '21:00:00+1:00'.

Example b:

```
TIME '12:00:00-5:00' AT TIME ZONE INTERVAL '1:00' HOUR TO MINUTE
```

Because the operand has a time zone, the result is independent of the session time zone displacement. Step 1 results in TIME '17:00:00+0:00', and step 2 results in TIME '18:00:00+1:00'

Note that the operand is not limited to datetime literals used in these examples. Any valid expression that evaluates to a datetime value can be the operand.

Type Conversion

CAST is used to for all other conversions. Examples:

```
CAST (<value> AS TIME WITHOUT TIME ZONE)
CAST (<value> AS TIME WITH TIME ZONE)
```

In the first example, if `<value>` has a time zone component, it is simply dropped. For example TIME '12:00:00-5:00' is converted to TIME '12:00:00'

In the second example, if `<value>` has no time zone component, the current time zone displacement of the session is added. For example TIME '12:00:00' is converted to TIME '12:00:00-8:00' when the session time zone displacement is '-8:00'.

Conversion between DATE and TIMESTAMP is performed by removing the TIME component of a TIMESTAMP value or by setting the hour, minute and second fields to zero. TIMESTAMP '2008-08-08 20:08:08+8:00' becomes DATE '2008-08-08', while DATE '2008-08-22' becomes TIMESTAMP '2008-08-22 00:00:00'.

Conversion between TIME and TIMESTAMP is performed by removing the DATE field values of a TIMESTAMP value or by appending the fields of the TIME value to the fields of the current session date value.

Assignment

When a value is assigned to a datetime target, e.g., a value is used to update a row of a table, the type of the value must be the same as the target, but the WITH TIME ZONE or WITHOUT TIME ZONE characteristics can be different. If the types are not the same, an explicit CAST must be used to convert the value into the target type.

Comparison

When values WITH TIME ZONE are compared, they are converted to UTC values before comparison. If a value WITH TIME ZONE is compared to another WITHOUT TIME ZONE, then the WITH TIME ZONE value is converted to AT LOCAL, then converted to WITHOUT TIME ZONE before comparison.

It is not recommended to design applications that rely on comparisons and conversions between TIME values WITH TIME ZONE. The conversions may involve normalisation of the time value, resulting in unexpected results. For example, the expression: BETWEEN(TIME '12:00:00-8:00', TIME '22:00:00-8:00') is converted to BETWEEN(TIME '20:00:00+0:00', TIME '06:00:00+0:00') when it is evaluated in the UTC zone, which is always FALSE.

Functions

Several functions return the current session timestamp in different datetime types:

CURRENT_DATE	DATE

CURRENT_TIME	TIME WITH TIME ZONE
CURRENT_TIMESTAMP	TIMESTAMP WITH TIME ZONE
LOCALTIME	TIMESTAMP WITHOUT TIME ZONE
LOCALTIMESTAMP	TIMESTAMP WITHOUT TIME ZONE

Session Time Zone Displacement

When an SQL session is started (with a JDBC connection) the local time zone of the client JVM (including any seasonal time adjustments such as daylight saving time) is used as the session time zone displacement. Note that the SQL session time displacement is not changed when a seasonal time adjustment takes place while the session is open. To change the SQL session time zone displacement use the following commands:

```
SET TIME ZONE <time displacement>

SET TIME ZONE LOCAL
```

The first command sets the displacement to the given value. The second command restores the original, real time zone displacement of the session.

Datetime Values and Java

When datetime values are sent to the database using the `PreparedStatement` or `CallableStatement` interfaces, the Java object is converted to the type of the prepared or callable statement parameter. This type may be DATE, TIME, or TIMESTAMP (with or without time zone). The time zone displacement is the time zone of the JDBC session.

When datetime values are retrieved from the database using the `ResultSet` interface, there are two representations. The `getString(…)` methods of the `ResultSet` interface, return an exact representation of the value in the SQL type as it is stored in the database. This includes the correct number of digits for the fractional second field, and for values with time zone displacement, the time zone displacement. Therefore if TIME '12:00:00' is stored in the database, all users in different time zones will get '12:00:00' when they retrieve the value as a string. The `getTime(…)` and `getTimestamp(…)` methods of the `ResultSet` interface return Java objects that are corrected for the session time zone. The UTC millisecond value contained the `java.sql.Time` or `java.sql.Timestamp` objects will be adjusted to the time zone of the session, therefore the `toString()` method of these objects return the same values in different time zones.

If you want to store and retrieve UTC values that are independent of any session's time zone, you can use a TIMESTAMP WITH TIME ZONE column. The setTime(...) and setTimestamp(...) methods of the PreparedStatement interface which have a Calendar parameter can be used to assign the values. The time zone of the given Calendar argument is used as the time zone. Conversely, the getTime(...) and getTimestamp(...) methods of the ResultSet interface which have a Calendar parameter can be used with a Calendar argument to retrieve the values.

JDBC has an unfortunate limitation and does not include type codes for SQL datetime types that have a TIME ZONE property. Therefore, for compatibility with database tools that are limited to the JDBC type codes, HyperSQL reports these types by default as datetime types without TIME ZONE. You can use the URL property `hsqldb.translate_dti_types=false` to override the default behaviour.

Interval Types

Interval types are used to represent differences between date time values. The difference between two date time values can be measured in seconds or in months. For measurements in months, the units YEAR and MONTH are available, while for measurements in seconds, the units DAY, HOUR, MINUTE, SECOND are available. The units can be used individually, or as a range. An interval type can specify the precision of the most significant field and the second fraction digits of the SECOND field (if it has a SECOND field). The default precision is 2. The default second precision is 0.

```
<interval type> ::= INTERVAL <interval qualifier>

<interval qualifier> ::= <start field> TO <end field> | <single datetime field>

<start field> ::= <non-second primary datetime field> [ <left paren> <interval
leading field precision> <right paren> ]

<end field> ::= <non-second primary datetime field> | SECOND [ <left paren>
<interval fractional seconds precision> <right paren> ]

<single datetime field> ::= <non-second primary datetime field> [ <left paren>
<interval leading field precision> <right paren> ] | SECOND [ <left paren>
<interval leading field precision> [ <comma> <interval fractional seconds pre-
cision> ] <right paren> ]

<primary datetime field> ::= <non-second primary datetime field> | SECOND

<non-second primary datetime field> ::= YEAR | MONTH | DAY | HOUR | MINUTE

<interval fractional seconds precision> ::= <unsigned integer>

<interval leading field precision> ::= <unsigned integer>
```

Examples of INTERVAL type definition:

```
INTERVAL YEAR TO MONTH
INTERVAL YEAR(3)
INTERVAL DAY(4) TO HOUR
INTERVAL MINUTE(4) TO SECOND(6)
INTERVAL SECOND(4,6)
```

The word INTERVAL indicates the general type name. The rest of the definition is called an `<interval quali-fier>`. This designation is important, as in most expressions `<interval qualifier>` is used without the word INTERVAL.

Interval Values

An interval value can be negative, positive or zero. An interval type has all the datetime fields in the specified range. These fields are similar to those in the TIMESTAMP type. The differences are as follows:

The first field of an interval value can hold any numeric value up to the specified precision. For example, the hour field in HOUR(2) TO SECOND can hold values above 23 (up to 99). The year and month fields can hold zero (unlike a TIMESTAMP value) and the maximum value of a month field that is not the most significant field, is 11.

The standard function `ABS(<interval value expression>)` can be used to convert a negative interval value to a positive one.

The literal representation of interval values consists of the type definition, with a string representing the interval value inserted after the word INTERVAL. Some examples of interval literal below:

```
INTERVAL '145 23:12:19.345' DAY(3) TO SECOND(3)
INTERVAL '3503:12:19.345' HOUR TO SECOND(3) /* equal to the first value */
INTERVAL '19.345' SECOND(4,3) /* maximum number of digits for the second value is 4, and each
 value is expressed with three fraction digits. */
INTERVAL '-23-10' YEAR(2) TO MONTH
```

Interval values of the types that are based on seconds can be cast into one another. Similarly those that are based on months can be cast into one another. It is not possible to cast or convert a value based on seconds to one based on months, or vice versa.

When a cast is performed to a type with a smaller least-significant field, nothing is lost from the interval value. Otherwise, the values for the missing least-significant fields are discarded. Examples:

```
CAST ( INTERVAL '145 23:12:19' DAY TO SECOND AS INTERVAL DAY TO HOUR ) = INTERVAL '145 23' DAY TO
 HOUR
CAST(INTERVAL '145 23' DAY TO HOUR AS INTERVAL DAY TO SECOND) = INTERVAL '145 23:00:00' DAY TO
 SECOND
```

A numeric value can be cast to an interval type. In this case the numeric value is first converted to a single-field INTERVAL type with the same field as the least significant field of the target interval type. This value is then converted to the target interval type For example CAST(22 AS INTERVAL YEAR TO MONTH) evaluates to INTERVAL '22' MONTH and then INTERVAL '1 10' YEAR TO MONTH. Note that SQL Standard only supports casts to single-field INTERVAL types, while HyperSQL allows casting to multi-field types as well.

An interval value can be cast to a numeric type. In this case the interval value is first converted to a single-field INTERVAL type with the same field as the least significant filed of the interval value. The value is then converted to the target type. For example CAST (INTERVAL '1-11' YEAR TO MONTH AS INT) evaluates to INTERVAL '23' MONTH, and then 23.

An interval value can be cast into a character type, which results in an INTERVAL literal. A character value can be cast into an INTERVAL type so long as it is a string with a format compatible with an INTERVAL literal.

Two interval values can be added or subtracted so long as the types of both are based on the same field, i.e., both are based on MONTH or SECOND. The values are both converted to a single-field interval type with same field as the least-significant field between the two types. After addition or subtraction, the result is converted to an interval type that contains all the fields of the two original types.

An interval value can be multiplied or divided by a numeric value. Again, the value is converted to a numeric, which is then multiplied or divided, before converting back to the original interval type.

An interval value is negated by simply prefixing with the minus sign.

Interval values used in expressions are either typed values, including interval literals, or are interval casts. The expression: `<expression> <interval qualifier>` is a cast of the result of the `<expression>` into the INTERVAL type specified by the `<interval qualifier>`. The cast can be formed by adding the keywords and parentheses as follows: CAST (`<expression>` AS INTERVAL `<interval qualifier>`).

The examples below feature different forms of expression that represent an interval value, which is then added to the given date literal.

```
DATE '2000-01-01' + INTERVAL '1-10' YEAR TO MONTH /* interval literal */
DATE '2000-01-01' + '1-10' YEAR TO MONTH /* the string '1-10' is cast into INTERVAL YEAR TO MONTH
 */
DATE '2000-01-01' + 22 MONTH /* the integer 22 is cast into INTERVAL MONTH, same value as above
 */
DATE '2000-01-01' - 22 DAY /* the integer 22 is cast into INTERVAL DAY */
DATE '2000-01-01' + COL2 /* the type of COL2 must be an INTERVAL type */
DATE '2000-01-01' + COL2 MONTH /* COL2 may be a number, it is cast into a MONTH interval */
```

Datetime and Interval Operations

An interval can be added to or subtracted from a datetime value so long as they have some fields in common. For example, an INTERVAL MONTH cannot be added to a TIME value, while an INTERVAL HOUR TO SECOND can. The interval is first converted to a numeric value, then the value is added to, or subtracted from, the corresponding field of the datetime value.

If the result of addition or subtraction is beyond the permissible range for the field, the field value is normalised and carried over to the next significant field until all the fields are normalised. For example, adding 20 minutes to TIME

'23:50:10' will result successively in '23:70:10', '24:10:10' and finally TIME '00:10:10'. Subtracting 20 minutes from the result is performed as follows: '00:-10:10', '-1:50:10', finally TIME '23:50:10'. Note that if DATE or TIMESTAMP normalisation results in the YEAR field value out of the range (1,1000), then an exception condition is raised.

If an interval value based on MONTH is added to, or subtracted from a DATE or TIMESTAMP value, the result may have an invalid day (30 or 31) for the given result month. In this case an exception condition is raised.

The result of subtraction of two datetime expressions is an interval value. The two datetime expressions must be of the same type. The type of the interval value must be specified in the expression, using only the interval field names. The two datetime expressions are enclosed in parentheses, followed by the `<interval qualifier>` fields. In the first example below, COL1 and COL2 are of the same datetime type, and the result is evaluated in INTERVAL YEAR TO MONTH type.

```
(COL1 - COL2) YEAR TO MONTH /* the difference between two DATE or two TIEMSTAMP values in years
 and months */
(CURRENT_DATE - COL3) DAY /* the number of days between the value of COL3 and the current date */
(CURRENT_DATE - DATE '2000-01-01') YEAR TO MONTH /* the number of years and months since the
 beginning of this century */
CURRENT_DATE - 2 DAY /* the date of the day before yesterday */
(CURRENT_TIMESTAMP - TIMESTAMP '2009-01-01 00:00:00') DAY(4) TO SECOND(2) /* days to seconds
 since the given date */
```

The individual fields of both datetime and interval values can be extracted using the EXTRACT function. The same function can also be used to extract the time zone displacement fields of a datetime value.

```
EXTRACT ({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | TIMEZONE_HOUR |
TIMEZONE_MINUTE | DAY_OF_WEEK | WEEK_OF_YEAR } FROM {<datetime value> | <interval
value>})
```

The dichotomy between interval types based on seconds, and those based on months, stems from the fact that the different calendar months have different numbers of days. For example, the expression, "nine months and nine days since an event" is not exact when the date of the event is unknown. It can represent a period of around 284 days give or take one. SQL interval values are independent of any start or end dates or times. However, when they are added to or subtracted from certain date or timestamp values, the result may be invalid and cause an exception (e.g. adding one month to January 30 results in February 30, which is invalid).

JDBC has an unfortunate limitation and does not include type codes for SQL INTERVAL types. Therefore, for compatibility with database tools that are limited to the JDBC type codes, HyperSQL reports these types by default as VARCHAR. You can use the URL property `hsqldb.translate_dti_types=false` to override the default behaviour.

Arrays

Array are a powerful feature of SQL:2008 and can help solve many common problems. Arrays should not be used as a substitute for tables.

HyperSQL supports arrays of values according to the SQL:2008 Standard.

Elements of the array are either NULL, or of the same data type. It is possible to define arrays of all supported types, including the types covered in this chapter and user defined types, except LOB types. An SQL array is one dimensional and is addressed from position 1. An empty array can also be used, which has no element.

Arrays can be stored in the database, as well as being used as temporary containers of values for simplifying SQL statements. They facilitate data exchange between the SQL engine and the user's application.

The full range of supported syntax allows array to be created, used in SELECT or other statements, combined with rows of tables and used in routine calls.

Array Definition

The type of a table column, a routine parameter, a variable, or the return value of a function can be defined as an array.

```
<array type> ::= <data type> ARRAY [ <left bracket or trigraph> <maximum car-
dinality> <right bracket or trigraph> ]
```

The word ARRAY is added to any valid type definition except BLOB and CLOB type definitions. If the optional `<maximum cardinality>` is not used, the default value is 1024. The size of the array cannot be extended beyond maximum cardinality.

In the example below, the table contains a column of integer arrays and a column of varchar arrays. The VARCHAR array has an explicit maximum size of 10, which means each array can have between 0 and 10 elements. The INTEGER array has the default maximum size of 1024. The id column has a default clause with an empty array. The default clause can be defined only as DEFAULT NULL or DEFAULT ARRAY[] and does not allow arrays containing elements.

```
CREATE TABLE t (id INT PRIMARY KEY, scores INT ARRAY DEFAULT ARRAY[], names VARCHAR(20)
 ARRAY[10])
```

An array can be constructed from value expressions or a query expression.

```
<array value constructor by enumeration> ::= ARRAY <left bracket or trigraph>
<array element list> <right bracket or trigraph>
```

```
<array element list> ::= <value expression> [ { <comma> <value expression> }... ]
```

```
<array value constructor by query> ::= ARRAY <left paren> <query expression>
[ <order by clause> ] <right paren>
```

In the examples below, arrays are constructed from values, column references or variables, function calls, or query expressions.

```
ARRAY [ 1, 2, 3 ]
ARRAY [ 'HOT', 'COLD' ]
ARRAY [ var1, var2, CURRENT_DATE ]
ARRAY (SELECT lastname FROM namestable ORDER BY id)
```

Array Reference

The most common operations on an array element reference and assignment, which are used when reading or writing an element of the array. Unlike Java and many other languages, arrays are extended if an element is assigned to an index beyond the current length. This can result in gaps containing NULL elements. Array length cannot exceed the maximum cardinality.

Elements of all arrays, including those that are the result of function calls or other operations can be referenced for reading.

```
<array element reference> ::= <array value expression> <left bracket> <numeric
value expression> <right bracket>
```

Elements of arrays that are table columns or routine variables can be referenced for writing. This is done in a SET statement, either inside an UPDATE statement, or as a separate statement in the case of routine variables, OUT and INOUT parameters.

```
<target array element specification> ::= <target array reference> <left bracket
or trigraph> <simple value specification> <right bracket or trigraph>
```

```
<target array reference> ::= <SQL parameter reference> | <column reference>
```

Note that only simple values or variables are allowed for the array index when an assignment is performed. The examples below demonstrates how elements of the array are referenced in SELECT and an UPDATE statement.

```
SELECT scores[ranking], names[ranking] FROM t JOIN t1 on (t.id = t1.tid)
UPDATE t SET scores[2] = 123, names[2] = 'Reds' WHERE id = 10
```

Array Operations

Several SQL operations and functions can be used with arrays.

CONCATENATION

Array concatenation is performed similar to string concatenation. All elements of the array on the right are appended to the array on left.

```
<array concatenation> ::= <array value expression 1> <concatenation operator>
<array value expression 2>
```

```
<concatenation operator> ::= ||
```

FUNCTIONS

Three functions operate on arrays. Details are described in the Built In Functions chapter.

```
CARDINALITY <left paren> <array value expression> <right paren>
```

```
MAX_CARDINALITY <left paren> <array value expression> <right paren>
```

Array cardinality and max cardinality are functions that return an integer. CARDINALITY returns the element count, while MAX_CARDINALITY returns the maximum declared cardinality of an array.

```
TRIM_ARRAY <left paren> <array value expression> <comma> <numeric value expres-
sion> <right paren>
```

The TRIM_ARRAY function returns a copy of an array with the specified number of elements removed from the end of the array. The `<array value expression>` can be any expression that evaluates to an array.

CAST

An array can be cast into an array of a different type. Each element of the array is cast into the element type of the target array type.

UNNEST

Arrays can be converted into table references with the UNNEST keyword.

```
UNNEST(<array value expression>) [ WITH ORDINALITY ]
```

The `<array value expression>` can be any expression that evaluates to an array. A table is returned that contains one column when WITH ORDINALITY is not used, or two columns when WITH ORDINALITY is used. The first column contains the elements of the array (including all the nulls). When the table has two columns, the second column contains the ordinal position of the element in the array. When UNNEST is used in the FROM clause of a query, it implies the LATERAL keyword, which means the array that is converted to table can belong to any table that precedes the UNNEST in the FROM clause. This is explained in the Data Access and Change chapter.

COMPARISON

Arrays can be compared for equality, but they cannot be compared for ordering or ranges. Array expressions are therefore not allowed in an ORDER BY clause, or in a comparison expression such as GREATER THAN. Two arrays are equal if they have the same length and the values at each index position are either equal or both NULL.

USER DEFINED FUNCTIONS and PROCEDURES

Array parameters, variables and return values can be specified in user defined functions and procedures, including aggregate functions. An aggregate function can return an array that contains all the scalar values that have been aggregated. These capabilities allow a wider range of applications to be covered by user defined functions and easier data exchange between the engine and the user's application.

Indexes and Query Speed

HyperSQL supports PRIMARY KEY, UNIQUE and FOREIGN KEY constraints, which can span multiple columns.

The engine creates indexes internally to support PRIMARY KEY, UNIQUE and FOREIGN KEY constraints: a unique index is created for each PRIMARY KEY or UNIQUE constraint; an ordinary index is created for each FOREIGN KEY constraint.

HyperSQL allows defining indexes on single or multiple columns. You should not create duplicate user-defined indexes on the same column sets covered by constraints. This would result in unnecessary memory and speed overheads. See the discussion in the System Management and Deployment Issues chapter for more information.

Indexes are crucial for adequate query speed. When range or equality conditions are used e.g. `SELECT ... WHERE acol > 10 AND bcol = 0`, an index should exist on one of the columns that has a condition. In this example, the `bcol` column is the best candidate. HyperSQL always uses the best condition and index. If there are two indexes, one on acol, and another on bcol, it will choose the index on bcol.

Queries always return results whether indexes exist or not, but they return much faster when an index exists. As a rule of thumb, HSQLDB is capable of internal processing of queries at over 100,000 rows per second. Any query that runs into several seconds is clearly accessing thousands of rows. The query should be checked and indexes should be added to the relevant columns of the tables if necessary. The EXPLAIN PLAN <query> statement can be used to see which indexes are used to process the query.

When executing a DELETE or UPDATE statement, the engine needs to find the rows that are to be deleted or updated. If there is an index on one of the columns in the WHERE clause, it is often possible to start directly from the first candidate row. Otherwise all the rows of the table have to be examined.

Indexes are even more important in joins between multiple tables. `SELECT ... FROM t1 JOIN t2 ON t1.c1 = t2.c2` is performed by taking rows of t1 one by one and finding a matching row in t2. If there is no index on t2.c2 then for each row of t1, all the rows of t2 must be checked. Whereas with an index, a matching row can be found in a fraction of the time. If the query also has a condition on t1, e.g., `SELECT ... FROM t1 JOIN t2 ON t1.c1 = t2.c2 WHERE t1.c3 = 4` then an index on t1.c3 would eliminate the need for checking all the rows of t1 one by one, and will reduce query time to less than a millisecond per returned row. So if t1 and t2 each contain 10,000 rows, the query without indexes involves checking 100,000,000 row combinations. With an index on t2.c2, this is reduced to 10,000 row checks and index lookups. With the additional index on t2.c2, only about 4 rows are checked to get the first result row.

Note that in HSQLDB an index on multiple columns can be used internally as a non-unique index on the first column in the list. For example: `CONSTRAINT name1 UNIQUE (c1, c2, c3);` means there is the equivalent of `CREATE INDEX name2 ON atable(c1);`. So you do not need to specify an extra index if you require one on the first column of the list.

In HyperSQL 2.0, a multi-column index will speed up queries that contain joins or values on the first n columns of the index. You need NOT declare additional individual indexes on those columns unless you use queries that search only

on a subset of the columns. For example, rows of a table that has a PRIMARY KEY or UNIQUE constraint on three columns or simply an ordinary index on those columns can be found efficiently when values for all three columns, or the first two columns, or the first column, are specified in the WHERE clause. For example, `SELECT  ...  FROM t1 WHERE t1.c1 = 4 AND t1.c2 = 6 AND t1.c3 = 8` will use an index on `t1(c1,c2,c3)` if it exists.

A multi-column index will not speed up queries on the second or third column only. The first column must be specified in the JOIN .. ON or WHERE conditions.

Sometimes query speed depends on the order of the tables in the JOIN .. ON or FROM clauses. For example the second query below should be faster with large tables (provided there is an index on `TB.COL3`). The reason is that `TB.COL3` can be evaluated very quickly if it applies to the first table (and there is an index on `TB.COL3`):

```
(TB is a very large table with only a few rows where TB.COL3 = 4)

SELECT * FROM TA JOIN TB ON TA.COL1 = TB.COL2 AND TB.COL3 = 4;

SELECT * FROM TB JOIN TA ON TA.COL1 = TB.COL2 AND TB.COL3 = 4;
```

The general rule is to put first the table that has a narrowing condition on one of its columns.

HyperSQL features automatic, on-the-fly indexes for views and subselects that are used in a query.

Indexes have no effect on some LIKE conditions.

Indexes are used for ORDER BY clauses if the same index is used for selection and ordering of rows.

Query Processing and Optimisation

HyperSQL does not change the order of tables in a query in order to optimise processing. As mentioned in the previous section, the table that has a narrowing condition should be the first table in the query.

HyperSQL optimises queries to use indexes, for all types of range and equality conditions, including IS NULL and NOT NULL conditions. Conditions can be in join or WHERE clauses, including all types of joins.

In addition, HyperSQL will always use an index (if one exists) for IN conditions, whether constants, variable, or subqueries are used on the right hand side of the IN predicate.

HyperSQL can always use indexes when several conditions are combined with the AND operator, choosing a conditions which can use an index. This now extended to all equality conditions on multiple columns that are part of an index.

HyperSQL will also use indexes when several conditions are combined with the OR operator and each condition can use an index (each condition may use a different index). For example, if a huge table has two separate columns for first name and last name, and both columns are indexed, a query such as the following example will use the indexes and complete in a short time:

```
(TC is a very large table)

SELECT * FROM TC WHERE TC.FIRSTNAME = 'John' OR TC.LASTNAME = 'Smith' OR TC.LASTNAME =
'Williams'
```

HyperSQL optimises simple row count queries in the form of SELECT COUNT(*) FROM <table> and returns the result immediately (this optimisation does not take place in MVCC mode).

HyperSQL can use an index on a column for SELECT MAX(<column>) FROM <table> and SELECT MIN(<column>) FROM <table> queries. There should be an index on the <column> and the query can have a WHERE condition on the same column. In the example below the maximum value for the TB.COL3 below 1000000 is returned.

```
SELECT MAX(TB.COL3) FROM TB WHERE TB.COL < 1000000
```

HyperSQL can use an index on an ORDER BY clause if all the columns in ORDER BY are in a single-column or multi-column index (in the exact order). This is important if there is a LIMIT n (or FETCH n ROWS ONLY) clause. In this situation, the use of index allows the query processor to access only the number of rows specified in the LIMIT clause, instead of building the whole result set, which can be huge. This also works for joined tables when the ORDER BY clause is on the columns of the first table in a join. Indexes are used in the same way when ORDER BY ... DESC is specified in the query. Note that unlike other RDBMS, HyperSQL does not create DESC indexes. It can use any index for ORDER BY ... DESC.

If there is an equality or range condition (e.g. EQUALS, GREATER THAN) condition on the columns specified in the ORDER BY clause, the index is still used. But if the query contains an equality condition on another indexed column in the table, this may take precedence and no index may be used for ORDER BY.

In the two examples below, the index on TB.COL3 is used and only up to 1000 rows are processed and returned.

```
   (TB is a very large table with an index on TB.COL3

   SELECT * FROM TB JOIN TA ON TA.COL1 = TB.COL2 WHERE TB.COL3 > 40000 ORDER BY TB.COL3 LIMIT
1000;
   SELECT * FROM TB JOIN TA ON TA.COL1 = TB.COL2 WHERE TB.COL3 > 40000 AND TB.COL3 < 100000
ORDER BY TB.COL3 DESC LIMIT 1000;
```

Chapter 3. Sessions and Transactions

Fred Toussi, The HSQL Development Group

Overview

All SQL statements are executed in sessions. When a connection is established to the database, a session is started. The authorization of the session is the name of the user that started the session. A session has several properties. These properties are set by default at the start according to database settings.

SQL Statements are generally transactional statements. When a transactional statement is executed, it starts a transaction if no transaction is in progress. If SQL Data is modified during a transaction, the change can be undone with a ROLLBACK statement. When a COMMIT statement is executed, the transaction is ended. If a single statement fails, the transaction is not normally terminated. However, some failures are caused by execution of statements that are in conflict with statements executed in other concurrent sessions. Such failures result in an implicit ROLLBACK, in addition to the exception that is raised.

Schema definition and manipulation statements are also transactional according to the SQL Standard. HyperSQL 2.0 performs automatic commits before and after the execution of such transactions. Therefore, schema-related statements cannot be rolled back. This is likely to change in future versions.

Some statements are not transactional. Most of these statements are used to change the properties of the session. These statements begin with the SET keyword.

If the AUTOCOMMIT property of a session is TRUE, then each transactional statement is followed by an implicit COMMIT.

The default isolation level for a session is READ COMMITTED. This can be changed using the JDBC `java.sql.Connection` object and its `setTransactionIsolation(int level)` method. The session can be put in read-only mode using the `setReadOnly(boolean readOnly)` method. Both methods can be invoked only after a commit or a rollback, but not during a transaction.

The isolation level and / or the readonly mode of a transaction can also be modified using an SQL statement. You can use the statement to change only the isolation mode, only the read-only mode, or both at the same time. This command can be issued only after a commit or rollback.

```
SET TRANSACTION <transaction characteristic> [ <comma> <transaction character-
istic> ]
```

Details of the statement is described later in this chapter.

Session Attributes and Variables

Each session has several system attributes. A session can also have user-defined session variables.

Session Attributes

The system attributes reflect the current mode of operation for the session. These attributes can be accessed with function calls and can be referenced in queries. For example, they can be returned using the `VALUES <attribute function>, ...` statement.

The named attributes such as CURRENT_USER, CURRENT_SCHEMA, etc. are SQL Standard functions. Other attributes of the session, such as auto-commit or read-only modes can be read using other built-in functions. All these functions are listed in the Built In Functions chapter.

Session Variables

Session variables are user-defined variables created the same way as the variables for stored procedures and functions. Currently, these variables cannot be used in general SQL statements. They can be assigned to IN, INOUT and OUT parameters of stored procedures. This allows calling stored procedures which have INOUT or OUT arguments and is useful for development and debugging. See the example in the SQL-Invoked Routines chapter, under Formal Parameters.

Example 3.1. User-defined Session Variables

```
DECLARE counter INTEGER DEFAULT 3;
DECLARE result VARCHAR(20) DEFAULT NULL;
SET counter=15;
CALL myroutine(counter, result)
```

Session Tables

With necessary access privileges, sessions can access all table, including GLOBAL TEMPORARY tables, that are defined in schemas. Although GLOBAL TEMPORARY tables have a single name and definition which applies to all sessions that use them, the contents of the tables are different for each session. The contents are cleared either at the end of each transaction or when the session is closed.

Session tables are different because their definition is visible only within the session that defines a table. The definition is dropped when the session is closed. Session tables do not belong to schemas.

```
<temporary table declaration> ::= DECLARE LOCAL TEMPORARY TABLE <table name>
<table element list> [ ON COMMIT { PRESERVE | DELETE } ROWS ]
```

The syntax for declaration is based on the SQL Standard. A session table cannot have FOREIGN KEY constraints, but it can have PRIMARY KEY, UNIQUE or CHECK constraints. A session table definition cannot be modified by adding or removing columns, indexes, etc.

It is possible to refer to a session table using its name, which takes precedence over a schema table of the same name. To distinguish a session table from schema tables, the pseudo schema name, MODULE can be used. An example is given below:

Example 3.2. User-defined Temporary Session Tables

```
 DECLARE LOCAL TEMPORARY TABLE buffer (id INTEGER PRIMARY KEY, textdata VARCHAR(100)) ON COMMIT
PRESERVE ROWS
 INSERT INTO module.buffer SELECT id, firstname || ' ' || lastname FROM customers
 -- do some more work
 DROP TABLE module.buffer
```

Session tables can be created inside a transaction. Automatic indexes are created and used on session tables when necessary for a query or other statement. By default, session table data is held in memory. If the session property

Transactions and Concurrency Control

HyperSQL 2.0 has been fully redesigned to support different transaction isolation models. It no longer supports the old 1.8.x model with "dirty read". Although it is perfectly possible to add an implementation of the transaction manager that supports the legacy model, we thought this is no longer necessary. The new system allows you to select the transaction isolation model even while the engine is running and choose different isolation modes for different simultaneous sessions.

HyperSQL 2.0 supports three concurrency control models, two-phase-locking (2PL), which is the default, multiversion concurrency control (MVCC) and a hybrid model, which is 2PL plus multiversion rows. Within each model, it supports some of 4 levels of transaction isolation: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE. The isolation level is a property of each SQL session, so different sessions can have different isolation levels. The concurrency control model is a strategy that governs all the sessions and is set for the database, as opposed for individual sessions. In the new implementation, all isolation levels avoid the "dirty read" phenomenon and do not read uncommitted changes made to rows by other transactions.

HyperSQL is fully multi threaded in all transaction models. Sessions continue to work simultaneously and can fully utilise multi-core processors.

To change the concurrency control model, the `SET DATABASE TRANSACTION CONTROL { LOCKS | MVLOCKS | MVCC }` can be used by a user with the DBA role.

Two Phase Locking

The two-phase locking model is the default mode. It is referred to by the keyword, LOCKS. In the 2PL model, each table that is read by a transaction is locked with a shared lock, and each table that is written to is locked with an exclusive lock. If two sessions read and modify different tables then both go through simultaneously. If one session tries to lock a table that has been locked by the other, if both locks are shared locks, it will go ahead. If either of the locks is an exclusive lock, the engine will put the session in wait until the other session commits or rolls back its transaction. In some cases the engine will invalidate the transaction of the current session, if the action would result in deadlock.

HyperSQL also supports explicit locking of a group of tables for the duration of the current transaction. Use of this command blocks access to the locked tables by other sessions and ensures the current session can complete the intended reads and writes on the locked tables.

If a table is read-only, it will not be locked by any transaction.

The READ UNCOMMITTED isolation level can be used in 2PL modes for read-only operations. It is the same as READ COMMITTED plus read only.

The READ COMMITTED isolation level is the default. It keeps write locks on tables until commit, but releases the read locks after each operation.

The REPEATABLE READ level is upgraded to SERIALIZABLE. These levels keep both read and write locks on tables until commit.

It is possible to perform some critical operations at the SERIALIZABLE level, while the rest of the operations are performed at the READ COMMITTED level.

Note: two phase locking refers to two periods in the life of a transaction. In the first period, locks are acquired, in the second period locks are released. No new lock is acquired after releasing a lock.

Two Phase Locking with Snapshot Isolation

This model is referred to as MVLOCKS. It works the same way as normal 2PL as far as updates are concerned.

SNAPSHOT ISOLATION is a multiversion concurrency strategy which uses the snapshot of the whole database at the time of the start of the transaction. In this model, read only transactions use SNAPSHOT ISOLATION. While other sessions are busy changing the database, the read only session sees a consistent view of the database and can access all the tables even when they are locked by other sessions for updates.

There are many applications for this mode of operation. In heavily updated data sets, this mode allows uninterrupted read access to the data.

Lock Contention in 2PL

When multiple connections are used to access the database, the transaction manager controls their activities. When each transaction performs only reads or writes on a single table, there is no contention. Each transaction waits until it can obtain a lock then performs the operation and commits. All contentions occur when transactions perform reads and writes on more than one table, or perform a read, followed by a write, on the same table.

For example, when sessions are working at the SERIALIZABLE level, when multiple sessions first read from a table in order to check if a row exists, then insert a row into the same table when it doesn't exist, there will be regular contention. Transaction A reads from the table, then does Transaction B. Now if either Transaction A or B attempts to insert a row, it will have to be terminated as the other transaction holds a shared lock on the table. If instead of two operations, a single MERGE statement is used to perform the read and write, no contention occurs because both locks are obtained at the same time.

Alternatively, there is the option of obtaining the necessary locks with an explicit LOCK TABLE statement. This statement should be executed before other statements and should include the names of all the tables and the locks needed. After this statement, all the other statements in the transaction can be executed and the transaction committed. The commit will remove all the locks.

HyperSQL is fully multi threaded. It therefore allows different transactions to execute concurrently so long as they are not modifying the same table.

MVCC

In the MVCC model, there are no shared, read locks. Exclusive locks are used on individual rows, but their use is different. Transactions can read and modify the same table simultaneously, generally without waiting for other transactions.

When transactions are running at READ COMMITTED level, no conflict will normally occur. If a transaction that runs at this level wants to modify a row that has been modified by another uncommitted transaction, then the engine puts the transaction in wait, until the other transaction has committed. The transaction then continues automatically. (Conflict is possible if each transaction is waiting for a different row modified by the other transaction, in which case, one of the transactions is terminated). This isolation level is called READ CONSISTENCY.

When transactions are running in REPEATABLE READ or SERIALIZABLE isolation levels, conflict is more likely to happen. There is no difference in operation between these two isolation levels. If a transaction that runs at these levels wants to modify a row that has been modified by another uncommitted transaction, the engine will invalidate the current transaction and roll back all its changes. This isolation level is called SNAPSHOT ISOLATION.

In the MVCC model, READ UNCOMMITTED is promoted to READ COMMITTED, as the new architecture is based on multi-version rows for uncommitted data and more than one version may exist for some rows.

With MVCC, when a transaction only reads data, then it will go ahead and complete regardless of what other transactions may do. This does not depend on the transaction being read-only or the isolation modes.

Choosing the Transaction Model

The SQL Standard defines the isolation levels as modes of operation that avoid the three unwanted phenomena, "dirty read", "fuzzy read" and "phantom row". The "dirty read" phenomenon occurs when a session can read a row that has been changed by another session. The "fuzzy read" phenomenon occurs when a row that was read by a session is modified by another session, then the first session reads the row again. The "phantom row" phenomenon occurs when a session performs an operation that affects several rows, for example, counts the rows or modifies them using a search condition, then another session adds one or more rows that fulfil the same search condition, then the first session performs an operation that relies on the results of its last operation. According to the Standard, the SERIALIZABLE isolation level avoids all three phenomena and also ensures that all the changes performed during a transaction can be considered as a series of uninterrupted changes to the database without any other transaction changing the database at all for the duration of these actions. The changes made by other transactions are considered to occur before the SERIALIZABLE transaction starts, or after it ends. The READ COMMITTED level avoids "dirty read" only, while the REPEATABLE READ level avoids "dirty read" and "fuzzy read", but not "phantom row".

The Standard allows the engine to return a higher isolation level than requested by the application. HyperSQL promotes a READ UNCOMMITTED request to READ COMMITTED and promotes a REPEATABLE READ request to SERIALIZABLE.

The MVCC model is not covered directly by the Standard. Research has established that the READ CONSISTENCY level fulfills the requirements of (and is stronger than) the READ COMMITTED level. The SNAPSHOT ISOLATION level is stronger than the READ CONSISTENCY level. It avoids the three anomalies defined by the Standard, and is therefore stronger than the REPEATABLE READ level as defined by the Standard. When operating with the MVCC model, HyperSQL treats a REPEATABLE READ or SERIALIZABLE setting for a transaction as SNAPSHOT ISOLATION.

All modes can be used with as many simultaneous connections as required. The default 2PL model is fine for applications with a single connection, or applications that do not access the same tables heavily for writes. With multiple simultaneous connections, MVCC can be used for most applications. Both READ CONSISTENCY and SNAPSHOT ISOLATION levels are stronger than the corresponding READ COMMITTED level in the 2PL mode. Some applications require SERIALIZABLE transactions for at least some of their operations. For these applications, one of the 2PL modes can be used. It is possible to switch the concurrency model while the database is operational. Therefore, the model can be changed for the duration of some special operations, such as synchronization with another data source.

All concurrency models are very fast in operation. When operating mainly on the same tables, the MVCC model may be faster with multiple processors.

Schema and Database Change

There are a few SQL statements that must access a consistent state of the database during their executions. These statements, which include CHECKPOINT and BACKUP, put an exclusive lock on all the tables of the database when they start.

Some schema manipulation statements put an exclusive lock on one or more tables. For example changing the columns of a table locks the table exclusively.

In the MVCC model, all statements that need an exclusive lock on one or more tables, put an exclusive lock on the database catalog until they complete.

The effect of these exclusive locks is similar to the execution of data manipulation statements with write locks. The session that is about to execute the schema change statement waits until no other session is holding a lock on any of the objects. At this point it starts its operation and locks the objects to prevents any other session from accessing the locked objects. As soon as the operation is complete, the locks are all removed.

Simultaneous Access to Tables

It was mentioned that there is no limit on the number of sessions that can access the tables and all sessions work simultaneously in multi threaded execution. However there are internal resources that are shared. Simultaneous access to these resources reduces the overall efficiency of the system. MEMORY and TEXT tables do not share resources and do not block multi threaded access. With CACHED tables, each write operation blocks the file and its cache until the operation is finished. With CACHED tables, SELECT operations do not block each other, but selecting from different tables and different parts of a large table causes the row cache to be updated frequently and will reduce overall performance.

The new access pattern is the opposite of the access pattern of version 1.8.x. In the old version, even when 20 sessions are actively reading and writing, only a single session at a time performs an SQL statement completely, before the next session is allowed access. In the new version, while a session is performing a SELECT statement and reading rows of a CACHED table to build a result set, another session may perform an UPDATE statement that reads and writes rows of the same table. The two operations are performed without any conflict, but the row cache is updated more frequently than when one operation is performed after the other operation has finished.

Session and Transaction Control Statements

SET AUTOCOMMIT

set autocommit command

```
<set autocommit statement> ::= SET AUTOCOMMIT { TRUE | FALSE }
```

When an SQL session is started by creating a JDBC connection, it is in AUTOCOMMIT mode. In this mode, after each SQL statement a COMMIT is performed automatically. This statement changes the mode. It is equivalent to using the `setAutoCommit( boolean autoCommit)` method of the JDBC `Connection` object.

START TRANSACTION

start transaction statement

```
<start transaction statement> ::= START TRANSACTION [ <transaction character-
istics> ]
```

Start an SQL transaction and set its characteristics. All transactional SQL statements start a transaction automatically, therefore using this statement is not necessary. If the statement is called in the middle of a transaction, an exception is thrown.

SET DATABASE TRANSACTION CONTROL

set database transaction control

```
<set database transaction control statement> ::= SET DATABASE TRANSACTION CONTROL
{ LOCKS | MVLOCKS | MVCC }
```

Set the concurrency control model for the whole database. It will wait until all sessions have been committed or rolled back. The default is LOCKS.

SET TRANSACTION

set next transaction characteristics

```
<set transaction statement> ::= SET [ LOCAL ] TRANSACTION <transaction char-
acteristics>
```

Set the characteristics of the next transaction in the current session. This statement has an effect only on the next transactions and has no effect on the future transactions after the next.

transaction characteristics

transaction characteristics

```
<transaction characteristics> ::= [ <transaction mode> [ { <comma> <transaction
mode> }... ] ]

<transaction mode> ::= <isolation level> | <transaction access mode> | <diag-
nostics size>

<transaction access mode> ::= READ ONLY | READ WRITE

<isolation level> ::= ISOLATION LEVEL <level of isolation>

<level of isolation> ::= READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ
| SERIALIZABLE

<diagnostics size> ::= DIAGNOSTICS SIZE <number of conditions>

<number of conditions> ::= <simple value specification>
```

Specify transaction characteristics.

Example 3.3. Setting Transaction Characteristics

```
SET TRANSACTION READ ONLY
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
SET TRANSACTION READ WRITE, ISOLATION LEVEL READ COMMITTED
```

SET CONSTRAINTS

set constraints mode statement

```
<set constraints mode statement> ::= SET CONSTRAINTS <constraint name list>
{ DEFERRED | IMMEDIATE }

<constraint name list> ::= ALL | <constraint name> [ { <comma> <constraint
name> }... ]
```

If the statement is issued during a transaction, it applies to the rest of the current transaction. If the statement is issued when a transaction is not active then it applies only to the next transaction in the current session. HyperSQL does not yet support this feature.

LOCK TABLE

lock table statement

```
<lock table statement> ::= LOCK TABLE <table name> { READ | WRITE} [, <table
name> { READ | WRITE} ...]}
```

In some circumstances, where multiple simultaneous transactions are in progress, it may be necessary to ensure a transaction consisting of several statements is completed, without being terminated due to possible deadlock. When this statement is executed, it waits until it can obtain all the listed locks, then returns. The SQL statements following this statements use the locks already obtained (and obtain new locks if necessary) and can proceed without waiting.

All the locks are released when a COMMIT or ROLLBACK statement is issued. Currently, this command does not have any effect when the database transaction control model is MVCC.

Example 3.4. Locking Tables

```
LOCK TABLE table_a WRITE, table_b READ
```

SAVEPOINT

savepoint statement

```
<savepoint statement> ::= SAVEPOINT <savepoint specifier>
```

```
<savepoint specifier> ::= <savepoint name>
```

Establish a savepoint. This command is used during an SQL transaction. It establishes a milestone for the current transaction. The SAVEPOINT can be used at a later point in the transaction to rollback the transaction to the milestone.

RELEASE SAVEPOINT

release savepoint statement

```
<release savepoint statement> ::= RELEASE SAVEPOINT <savepoint specifier>
```

Destroy a savepoint. This command is rarely used as it is not very useful. It removes a SAVEPOINT that has already been defined.

COMMIT

commit statement

```
<commit statement> ::= COMMIT [ WORK ] [ AND [ NO ] CHAIN ]
```

Terminate the current SQL-transaction with commit. This make all the changes to the database permanent.

ROLLBACK

rollback statement

```
<rollback statement> ::= ROLLBACK [ WORK ] [ AND [ NO ] CHAIN ]
```

Rollback the current SQL transaction and terminate it. The statement rolls back all the actions performed during the transaction. If NO CHAIN is specified, a new SQL transaction is started just after the rollback. The new transaction inherits the properties of the old transaction.

ROLLBACK TO SAVEPOINT

rollback statement

```
<rollback statement> ::= ROLLBACK [ WORK ] TO SAVEPOINT <savepoint specifier>
```

Rollback part of the current SQL transaction and continue the transaction. The statement rolls back all the actions performed after the specified SAVEPOINT was created. The same effect can be achieved with the `rollback( Savepoint savepoint)` method of the JDBC `Connection` object.

Example 3.5. Rollback

```
-- perform some inserts, deletes, etc.
```

```
SAVEPOINT A
-- perform some inserts, deletes, selects etc.
ROLLBACK WORK TO SAVEPOINT A
-- all the work after the declaration of SAVEPOINT A is rolled back
```

DISCONNECT

disconnect statement

```
<disconnect statement> ::= DISCONNECT
```

Terminate the current SQL session. Closing a JDBC connection has the same effect as this command.

SET SESSION CHARACTERISTICS

set session characteristics statement

```
<set session characteristics statement> ::= SET SESSION CHARACTERISTICS AS
<session characteristic list>

<session characteristic list> ::= <session characteristic> [ { <comma> <session
characteristic> }... ]

<session characteristic> ::= <session transaction characteristics>

<session transaction characteristics> ::= TRANSACTION <transaction mode>
[ { <comma> <transaction mode> }... ]
```

Set one or more characteristics for the current SQL-session. This command is used to set the transaction mode for the session. This endures for all transactions until the session is closed or the next use of this command. The current read-only mode can be accessed with the ISREADONLY() function.

Example 3.6. Setting Session Characteristics

```
SET SESSION CHARACTERISTICS AS TRANSACTION READ ONLY
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE
SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE, ISOLATION LEVEL READ COMMITTED
```

SET SESSION AUTHORIZATION

set session user identifier statement

```
<set session user identifier statement> ::= SET SESSION AUTHORIZATION <value
specification>
```

Set the SQL-session user identifier. This statement changes the current user. The user that executes this command must have the CHANGE_AUTHORIZATION role, or the DBA role. After this statement is executed, all SQL statements are executed with the privileges of the new user. The current authorisation can be accessed with the CURRENT_USER and SESSION_USER functions.

Example 3.7. Setting Session Authorization

```
SET SESSION AUTHORIZATION 'FELIX'
SET SESSION AUTHORIZATION SESSION_USER
```

SET ROLE

set role statement

```
<set role statement> ::= SET ROLE <role specification>

<role specification> ::= <value specification> | NONE
```

Set the SQL-session role name and the current role name for the current SQL-session context. The user that executes this command must have the specified role. If NONE is specified, then the previous CURRENT_ROLE is eliminated. The effect of this lasts for the lifetime of the session. The current role can be accessed with the CURRENT_ROLE function.

SET TIME ZONE

set local time zone statement

```
<set local time zone statement> ::= SET TIME ZONE <set time zone value>

<set time zone value> ::= <interval value expression> | LOCAL
```

Set the current default time zone displacement for the current SQL-session. When the session starts, the time zone displacement is set to the time zone of the client. This command changes the time zone displacement. The effect of this lasts for the lifetime of the session. If LOCAL is specified, the time zone displacement reverts to the local time zone of the session.

Example 3.8. Setting Session Time Zone

```
    SET TIME ZONE LOCAL
    SET TIME ZONE INTERVAL '+6:00' HOUR TO MINUTE
```

SET CATALOG

set catalog statement

```
<set catalog statement> ::= SET <catalog name characteristic>

<catalog name characteristic> ::= CATALOG <value specification>
```

Set the default schema name for unqualified names used in SQL statements that are prepared or executed directly in the current sessions. As there is only one catalog in the database, only the name of this catalog can be used. The current catalog can be accessed with the CURRENT_CATALOG function.

SET SCHEMA

set schema statement

```
<set schema statement> ::= SET <schema name characteristic>

<schema name characteristic> ::= SCHEMA <value specification> | <schema name>
```

Set the default schema name for unqualified names used in SQL statements that are prepared or executed directly in the current sessions. The effect of this lasts for the lifetime of the session. The SQL Standard form requires the schema name as a single-quoted string. HyperSQL also allows the use of the identifier for the schema. The current schema can be accessed with the CURRENT_SCHEMA function.

SET PATH

set path statement

```
<set path statement> ::= SET <SQL-path characteristic>

<SQL-path characteristic> ::= PATH <value specification>
```

Set the SQL-path used to determine the subject routine of routine invocations with unqualified routine names used in SQL statements that are prepared or executed directly in the current sessions. The effect of this lasts for the lifetime of the session.

SET MAXROWS

set max rows statement

```
<set max rows statement> ::= SET MAXROWS <unsigned integer literal>
```

The normal operation of the session has no limit on the number of rows returned from a SELECT statement. This command set the maximum number of rows of the result returned by executing queries.

This statement has a similar effect to the `setMaxRows(int max)` method of the JDBC `Statement` interface, but it affects the results returned from the next statement execution only. After the execution of the next statement, the MAXROWS limit is removed.

Only zero or positive values can be used with this command. The value overrides any value specified with `setMaxRows(int max)` method of a JDBC statement. The statement `SET MAXROWS 0` means no limit.

It is possible to limit the number of rows returned from SELECT statements with the FETCH <n> ROWS ONLY, or its alternative, LIMIT <n>. Therefore this command is not recommended for general use. The only legitimate use of this command is for checking and testing queries that may return very large numbers of rows.

SET SESSION RESULT MEMORY ROWS

set session result memory rows statement

```
<set session result memory rows statement> ::= SET SESSION RESULT MEMORY ROWS
<unsigned integer literal>
```

By default the session uses memory to build result sets, subquery results and temporary tables. This command sets the maximum number of rows of the result (and temporary tables) that should be kept in memory. If the row count of the result or temporary table exceeds the setting, the result is stored on disk. The default is 0, meaning all result sets are held in memory.

This statement applies to the current session only. The general database setting is:

```
SET DATABASE DEFAULT RESULT MEMORY ROWS <unsigned integer literal>
```

SET IGNORECASE

set ignore case statement

```
<set ignore case statement> ::= SET IGNORECASE { TRUE | FALSE }
```

Sets the type used for new VARCHAR table columns. By default, character columns in new databases are case sensitive. If `SET IGNORECASE TRUE` is used, all VARCHAR columns in new tables are set to `VARCHAR_IGNORECASE`. It is possible to specify the `VARCHAR_IGNORECASE` type for the definition of individual columns. So it is possible to have some columns case sensitive and some not, even in the same table. This statement must be switched before creating tables. Existing tables and their data are not affected.

Chapter 4. Schemas and Database Objects

Fred Toussi, The HSQL Development Group

$Revision: 3622 $

Copyright 2009 Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2010-06-04 11:33:51 -0400 (Fri, 04 Jun 2010) $

Overview

The persistent elements of an SQL environment are database objects. The database consists of catalogs plus authorizations.

A catalog contains schemas, while schemas contain the objects that contain data or govern the data.

Each catalog contains a special schema called INFORMATION_SCHEMA. This schema is read-only and contains some views and other schema objects. The views contain lists of all the database objects that exist within the catalog, plus all authorizations.

Each database object has a name. A name is an identifier and is unique within its name-space.

Schemas and Schema Objects

In HyperSQL, there is only one catalog per database. The name of the catalog is PUBLIC. You can rename the catalog with the `ALTER CATALOG RENAME TO` statement. All schemas belong the this catalog. The catalog name has no relation to the file name of the database.

Each database has also an internal "unique" name which is automatically generated when the database is created. This name is used for event logging. You can also change this unique name.

Schema objects are database objects that contain data or govern or perform operations on data. By definition, each schema object belongs to a specific schema.

Schema objects can be divided into groups according to their characteristics.

- Some kinds of schema objects can exist independently from other schema object. Other kinds can exist only as an element of another schema object. These dependent objects are automatically destroyed when the parent object is dropped.

- Separate name-spaces exists for different kinds of schema object. Some name-spaces are shared between two similar kinds of schema objects.

- There can be dependencies between various schema objects, as a schema object can include references to other schema objects. These references can cross schema boundaries. Interdependence and cross referencing between schema objects is allowed in some circumstances and disallowed in some others.

- Schema objects can be destroyed with the DROP statement. If dependent schema objects exist, a DROP statement will succeed only if it has a CASCADE clause. In this case, dependent objects are also destroyed in most cases. In some cases, such as dropping DOMAIN objects, the dependent objects are not destroyed, but modified to remove the dependency.

A new HyperSQL catalog contains an empty schema called PUBLIC. By default, this schema is the initial schema when a new session is started. New schemas and schema objects can be defined and used in the PUBLIC schema, as well as any new schema that is created by the user. You can rename the PUBLIC schema.

HyperSQL allows all schemas to be dropped, except the schema that is the default initial schema for new sessions (by default, the PUBLIC schema). For this schema, a DROP SCHEMA ... CASCADE statement will succeed but will result in an empty schema, rather than no schema.

The statements for setting the initial schema for users are described in the Statements for Authorization and Access Control chapter.

Names and References

The name of a schema object is an `<identifier>`. The name belongs to the name-space for the particular kind of schema object. The name is unique within its name-space. For example, each schema has a separate name-space for TRIGGER objects.

In addition to the name-spaces in the schema. Each table has a name-space for the names of its columns.

Because a schema object is always in a schema and a schema always in a catalog, it is possible, and sometimes necessary, to qualify the name of the schema object that is being referenced in an SQL statement. This is done by forming an `<identifier chain>`. In some contexts, only a simple `<identifier>` can be used and the `<identifier chain>` is prohibited. While in some other contexts, the use of `<identifier chain>` is optional. An identifier chain is formed by qualifying each object with the name of the object that owns its name-space. Therefore a column name is prefixed with a table name, a table name is prefixed with a schema name, and a schema name is prefixed with a catalog name. A fully qualified column name is in the form `<catalog name>.<schema name>.<table name>.<column name>`, likewise, a fully qualified sequence name is in the form `<catalog name>.<schema name>.<sequence name>`.

HyperSQL extends the SQL standard to allow renaming all database objects. The ALTER ... RENAME TO command has slightly different forms depending on the type of object. If an object is referenced in a VIEW or ROUTINE definition, it is not always possible to rename it.

Character Sets

A CHARACTER SET is the whole or a subset of the UNICODE character set.

A character set name can only be a `<regular identifier>`. There is a separate name-space for character sets.

There are several predefined character sets. These character sets belong to INFORMATION_SCHEMA. However, when they are referenced in a statement, no schema prefix can be used in the statement that references them.

The following character sets have been specified by the SQL Standard:

SQL_TEXT, SQL_IDENTIFIER, SQL_CHARACTER, ASCII_GRAPHIC, GRAPHIC_IRV, ASCII_FULL, ISO8BIT, LATIN1, UTF32, UTF16, UTF8.

The ASCII_GRAPHIC is the same as GRAPHIC_IRV and ASCII_FULL is the same as ISO8BIT.

Most of the character sets are defined by well-known standards such as UNICODE.

The SQL_CHARACTER consists of ASCII letters, digits and the symbols used in the SQL language. The SQL_TEXT, SQL_IDENTIFIER are implementation defined. HyperSQL defines SQL_TEXT as the UNICODE character set and SQL_IDENTIFIER as the UNICODE character set minus the SQL language special characters.

The character repertoire of HyperSQL is the UTF16 character set, which covers all possible character sets. If a predefined character set is specified for a table column, then any string stored in the column must contain only characters from the specified character set.

Early releases of HyperSQL version 2.0 may not enforce the CHARACTER SET that is specified for a column and may accept any character string.

Collations

A COLLATION is the method used for ordering character strings in ordered sets and to determine equivalence of two character strings.

There are several predefined collations. These collations belong to INFORMATION_SCHEMA. However, when they are referenced in a statement, no schema prefix can be used in the statement that references them.

There is a separate name-space for collations..

Collations for a large number of languages are supported by HyperSQL.

Early releases of HyperSQL version 2.0 only support a single collation for the whole database.

Distinct Types

A distinct, user-defined TYPE is simply based on a built-in type. A distinct TYPE is used in table definitions and in CAST statements.

Distinct types share a name-space with domains.

Domains

A DOMAIN is a user-defined type, simply based on a built-in type. A DOMAIN can have constraints that limit the values that the DOMAIN can represent. A DOMAIN can be used in table definitions and in CAST statements.

Distinct types share a name-space with domains.

Number Sequences

A SEQUENCE object produces INTEGER values in sequence. The SEQUENCE can be referenced in special contexts only within certain SQL statements. For each row where the object is referenced, its value is incremented.

There is a separate name-space for SEQUENCE objects.

IDENTITY columns are columns of tables which have an internal, unnamed SEQUENCE object.

SEQUENCE objects and IDENTITY columns are supported fully according to the latest SQL 2008 Standard syntax.

Sequences

The SQL:2008 syntax and usage is different from what is supported by many existing database engines. Sequences are created with the `CREATE SEQUENCE` command and their current value can be modified at any time with `ALTER SEQUENCE`. The next value for a sequence is retrieved with the `NEXT VALUE FOR <name>` expression. This expression can be used for inserting and updating table rows.

Example 4.1. inserting the next sequence value into a table row

```
INSERT INTO mytable VALUES 2, 'John', NEXT VALUE FOR mysequence;
```

You can also use it in select statements. For example, if you want to number the returned rows of a SELECT in sequential order, you can use:

Example 4.2. numbering returned rows of a SELECT in sequential order

```
SELECT NEXT VALUE FOR mysequence, col1, col2 FROM mytable WHERE ...
```

In version 2.0, the semantics of sequences is exactly as defined by SQL:2008. If you use the same sequence twice in the same row in an INSERT statement, you will get the same value as required by the Standard.

The correct way to use a sequence value is the NEXT VALUE FOR expression. You can query the SEQUENCES table for the next value that will be returned from any of the defined sequences. The SEQUENCE_NAME column contains the name and the NEXT_VALUE column contains the next value to be returned. Note that this is only for getting information and you should not use the sequence value.

Identity Auto-Increment Columns

Each table can contain a single auto-increment column, known as the IDENTITY column. An IDENTITY column is a SMALLINT, INTEGER, BIGINT, DECIMAL or NUMERIC column with its value generated by a sequence generator.

In HyperSQL 2.0, an IDENTITY column is not by default treated as the primary key for the table (as a result, multi-column primary keys are possible with an IDENTITY column present).

The SQL standard syntax is used, which allows the initial value and other options to be specified.

```
<colname> [ INTEGER | BIGINT | DECIMAL | NUMERIC ] GENERATED { BY DEFAULT | ALWAYS} AS IDENTITY
 [( <options> )] [PRIMARY KEY]
```

When you add a new row to such a table using an `INSERT INTO <tablename> ...` statement, you can use the DEFAULT keyword for the IDENTITY column, which results in an auto-generated value for the column. The `IDENTITY()` function returns the last value inserted into any IDENTITY column by this session. Each session manages this function call separately and is not affected by inserts in other sessions. Use `CALL IDENTITY()` as an SQL statement to retrieve this value. If you want to use the value for a field in a child table, you can use `INSERT INTO <childtable> VALUES (...,IDENTITY(),...);`. Both types of call to `IDENTITY()` must be made before any additional update or insert statements are issued by the session.

The last inserted IDENTITY value can also be retrieved via JDBC, by specifying the Statement or PreparedStatement object to return the generated value.

The next IDENTITY value to be used can be changed with following statement. Note that this statement is not used in normal operation and is only for special purposes:

```
ALTER TABLE ALTER COLUMN <column name> RESTART WITH <new value>;
```

For backward compatibility, support has been retained for `CREATE TABLE <tablename>(<colname> IDENTITY, ...)` as a shortcut which defines the column both as an IDENTITY column and a PRIMARY KEY column. Also, for backward compatibility, it is possible to use NULL as the value of an IDENTITY column in an INSERT statement and the value will be generated automatically. You should avoid these compatibility features as they may be removed from future versions of HyperSQL.

In the following example, the identity value for the first INSERT statement is generated automatically using the DEFAULT keyword. The second INSERT statement uses a call to the IDENTITY() function to populate a row in the child table with the generated identity value.

```
CREATE TABLE star (id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    firstname VARCHAR(20),
    lastname VARCHAR(20))
CREATE TABLE movies (starid INTEGER, movieid INTEGER PRIMARY KEY, title VARCHAR(40))
```

```
INSERT INTO star (id, firstname, lastname) VALUES (DEFAULT, 'Felix', 'the Cat')
INSERT INTO movies (starid, movieid, title) VALUES (IDENTITY(), 10, 'Felix in Hollywood')
```

Tables

In the SQL environment, tables are the most essential components, as they hold all persistent data.

If TABLE is considered as metadata (i.e. without its actual data) it is called a *relation* in relational theory. It has one or more columns, with each column having a distinct name and a data type. A table usually has one or more constraints which limit the values that can potentially be stored in the TABLE. These constraints are discussed in the next section.

A single column of the table can be defined as IDENTITY. The values stored in this column are auto-generated and are based on an (unnamed) identity sequence.

Views

A VIEW is similar to a TABLE but it does not permanently contain rows of data. A view is defined as a QUERY EXPRESSION, which is often a SELECT statement that references views and tables, but it can also consist of a TABLE CONSTRUCTOR that does not reference any tables or views.

A view has many uses:

- Hide the structure and column names of tables. The view can represent one or more tables or views as a separate table. This can include aggregate data, such as sums and averages, from other tables.

- Allow access to specific rows in a table. For example, allow access to records that were added since a given date, while hiding older records.

- Allow access to specific columns. For example allow access to columns that contain non-confidential information. Note that this can also be achieved with the GRANT SELECT statement, using column-level privileges

A VIEW that returns the columns of a single ordinary TABLE may be *updatable*. Some *updatable* views are *insertable-into*. When rows of an updatable view are updated, or new rows are inserted, these changes are reflected in the base table. A VIEW definition may specify that the inserted or updated rows conform to the search condition of the view. This is done with the CHECK OPTION clause.

Views share a name-space with tables.

Constraints

A CONSTRAINT is a child schema object and can belong to a DOMAIN or a TABLE. CONSTRAINT objects can be defined without specifying a name. In this case the system generates a name for the new object beginning with "SYS_".

In a DOMAIN, CHECK constraints can be defined that limits the value represented by the DOMAIN. These constraints work exactly like a CHECK constraint on a single column of a table as described below.

In a TABLE, a constraint takes three basic forms.

CHECK

A CHECK constraint consists of a `<search condition>` that must not be false (can be unknown) for each row of the table. The `<search condition>` can reference all the columns of the current row, and if it contains a `<subquery>`, other tables and views in the database (excluding its own table).

NOT NULL

A simple form of check constraint is the NOT NULL constraint, which applies to a single column.

UNIQUE

A UNIQUE constraint is based on an equality comparison of values of specific columns (taken together) of one row with the same values from each of the other rows. The result of the comparison must never be true (can be false or unknown). If a row of the table has NULL in any of the columns of the constraint, it conforms to the constraint. A unique constraint on multiple columns (c1, c2, c3, ..) means that in no two rows, the sets of values for the columns can be equal unless at lease one of them is NULL. Each single column taken by itself can have repeat values in different rows. The following example satisfies a UNIQUE constraint on the two columns

Example 4.3. Column values which satisfy a 2-column UNIQUE constraint

```
1,                                     2
2,                                     1
2,                                     2
NULL,                                  1
NULL,                                  1
1,                                     NULL
NULL,                                  NULL
NULL,                                  NULL
```

PRIMARY KEY

A PRIMARY KEY constraint is equivalent to a UNIQUE constraint on one or more NOT NULL columns. Only one PRIMARY KEY can be defined in each table.

FOREIGN KEY

A FOREIGN key constraint is based on an equality comparison between values of specific columns (taken together) of each row with the values of the columns of a UNIQUE constraint on another table or the same table. The result of the comparison must never be false (can be unknown). A special form of FOREIGN KEY constraint, based on its CHECK clause, allows the result to be unknown only if the values for all columns are NULL. A FOREIGN key can be declared only if a UNIQUE constraint exists on the referenced columns.

Constraints share a name space with assertions.

Assertions

An ASSERTION is a top-level schema objects. It consists of a `<search condition>` that must not be false (can be unknown).

Assertions share a name-space with constraints

Triggers

A TRIGGER is a child schema object that always belongs to a TABLE or a VIEW.

Each time a DELETE, UPDATE or INSERT is performed on the table or view, additional actions are taken by the triggers that have been declared on the table or view.

Triggers are discussed in detail in chapter Triggers .

Routines

Routines are user-defined functions or procedures. The names and usage of functions and procedures are different. FUNCTION is a routine that can be referenced in many types of statements. PROCEDURE is a routine that can be referenced only in a CALL statement.

There is a separate name-space for routines.

Because of the possibility of overloading, each routine can have more than one name. The name of the routine is the same for all overloaded variants, but each variant has a *specific name*, different from all other routine names and specific names in the schema. The *specific name* can be specified in the routine definition statement. Otherwise it is assigned by the engine. The specific name is used only for schema manipulation statements, which need to reference a specific variant of the routine. For example, if a routine has two signatures, each signature has its own *specific name*. This allows the user to drop one of the signatures while keeping the other.

Routines are discussed in detail in chapter SQL-Invoked Routines .

Indexes

Indexes are an implementation-defined extension to the SQL Standard. HyperSQL has a dedicated name-space for indexes in each schema.

Statements for Schema Definition and Manipulation

Schemas and schema objects can be created, modified and dropped. The SQL Standard defines a range of statements for this purpose. HyperSQL supports many additional statements, especially for changing the properties of existing schema objects.

Common Elements and Statements

These elements and statements are used for different types of object. They are described here, before the statements that can use them.

identifier

definition of identifier

```
<identifier> ::= <regular identifier> | <delimited identifier> | <SQL language
identifier>

<delimited identifier> ::= <double quote> <character sequence> <double quote>

<regular identifier> ::= <special character sequence>

<SQL language identifier> ::= <special character sequence>
```

A `<delimited identifier>` is a sequence of characters enclosed with double-quote symbols. All characters are allowed in the character sequence.

A `<regular identifier>` is a special sequence of characters. It consists of letters, digits and the underscore characters. It must begin with a letter.

A `<SQL language identifier>` is similar to `<regular identifier>` but the letters can range only from A-Z in the ASCII character set. This type of identifier is used for names of CHARACTER SET objects.

If the character sequence of a delimited identifier is the same as an undelimited identifier, it represents the same identifier. For example "JOHN" is the same identifier as JOHN. In a `<regular identifier>` the case-normal form is considered for comparison. This form consists of the upper-case of equivalent of all the letters.

The character sequence length of all identifiers must be between 1 and 128 characters.

A reserved word is one that is used by the SQL Standard for special purposes. It is similar to a `<regular identi-fier>` but it cannot be used as an identifier for user objects. If a reserved word is enclosed in double quote characters, it becomes a quoted identifier and can be used for database objects.

CASCADE or RESTRICT

drop behavior

```
<drop behavior> ::= CASCADE | RESTRICT
```

The `<drop behavior>` is a required element of statements that drop a SCHEMA or a schema object. If `<drop behavior>` is not specified then `RESTRICT` is implicit. It determines the effect of the statement if there are other objects in the catalog that reference the SCHEMA or the schema object. If RESTRICT is specified, the statement fails if there are referencing objects. If CASCADE is specified, all the referencing objects are modified or dropped with cascading effect. Whether a referencing object is modified or dropped, depends on the kind of schema object that is dropped.

IF EXISTS

drop condition (HyperSQL)

```
<if exists clause> ::= IF EXISTS
```

This clause is not part of the SQL standard and is a HyperSQL extension to some commands that drop objects (schemas, tables, views, sequences and indexes). If it is specified, then the statement does not return an error if the drop statement is issued on a non-existent object.

SPECIFIC

specific routine designator

```
<specific routine designator> ::= SPECIFIC <routine type> <specific name>

<routine type> ::= ROUTINE | FUNCTION | PROCEDURE
```

This clause is used in statements that need to specify one of the multiple versions of an overloaded routine. The `<specific name>` is the one specified in the `<routine definition>` statement.

Renaming Objects

RENAME

rename statement (HyperSQL)

```
<rename statement> ::= ALTER <object type> <name> RENAME TO <new name>

<object type> ::= CATALOG | SCHEMA | DOMAIN | TYPE | TABLE | CONSTRAINT | INDEX
| ROUTINE | SPECIFIC ROUTINE

<column rename statement> ::= ALTER TABLE <table name> ALTER COLUMN <name>
RENAME TO <new name>
```

This statement is used to rename an existing object. It is not part of the SQL Standard. The specified `<name>` is the existing name, which can be qualified with a schema name, while the `<new name>` is the new name for the object.

Commenting Objects

COMMENT

comment statement (HyperSQL)

```
<comment statement> ::= COMMENT ON { TABLE | COLUMN | ROUTINE } <name> IS
<character string literal>
```

Adds a comment to the object metadata, which can later be read from an INFORMATION_SCHEMA view. This command is not part of the SQL Standard. The strange syntax is due to compatibility with other database engines that support the statement. The `<name>` is the name of a table, view, column or routine. The name of the column consists of dot-separated `<table name>` . `<column name>`. The name of the table, view or routine can be a simple name. All names can be qualified with a schema name. If there is alread a comment on the object, the new comment will replace it.

The comments appear in the results returned by JDBC DatabaseMetaData methods, getTables() and getColumns(). The INFORMATION_SCHEMA.SYSTEM_COMMENTS view contains the comments. You can query this view using the schema, table, and column names to retreive the comments.

Schema Creation

CREATE SCHEMA

schema definition

The CREATE_SCHEMA or DBA role is required in order to create a schema. A schema can be created with or without schema objects. Schema objects can always be added after creating the schema, or existing ones can be dropped. Within the `<schema definition>` statement, all schema object creation takes place inside the newly created schema. Therefore, if a schema name is specified for the schema objects, the name must match that of the new schema. In addition to statements for creating schema objects, the statement can include instances of `<grant statement>` and `<role definition>`. This is a curious aspect of the SQL standard, as these elements do not really belong to schema creation.

```
<schema definition> ::= CREATE SCHEMA <schema name clause> [ <schema character
set specification> ] [ <schema element>... ]

<schema name clause> ::= <schema name> | AUTHORIZATION <authorization identifier>
| <schema name> AUTHORIZATION <authorization identifier>
```

If the name of the schema is specified simply as `<schema name>`, then the AUTHORIZATION is the current user. Otherwise, the specified `<authorization identifier>` is used as the AUTHORIZATION for the schema. If `<schema name>` is omitted, then the name of the schema is the same as the specified `<authorization identifier>`.

```
<schema element> ::= <table definition> | <view definition> | <domain defini-
tion> | <character set definition> | <collation definition> | <transliteration
definition> | <assertion definition> | <trigger definition> | <user-defined type
definition> | <user-defined cast definition> | <user-defined ordering defini-
tion> | <transform definition> | <schema routine> | <sequence generator defi-
nition> | <grant statement> | <role definition>
```

An example of the command is given below. Note that a single semicolon appears at the end, there should be no semicolon between the statements:

```
    CREATE SCHEMA ACCOUNTS AUTHORIZATION DBA
        CREATE TABLE AB(A INTEGER, ...)
        CREATE TABLE CD(C CHAR(10), ...)
        CREATE VIEW VI AS SELECT ...
```

```
        GRANT SELECT ON AB TO PUBLIC
        GRANT SELECT ON CD TO JOE;
```

It is not really necessary to create a schema and all its objects as one command. The schema can be created first, and its objects can be created one by one.

DROP SCHEMA

drop schema statement

```
<drop schema statement> ::= DROP SCHEMA [ IF EXISTS ] <schema name> [ IF EXISTS ]
<drop behavior>
```

This command destroys an existing schema. If `<drop behavior>` is RESTRICT, the schema must be empty, otherwise an error is raised. If CASCADE is specified, then all the objects contained in the schema are destroyed with a CASCADE option.

Table Creation and Manipulation

CREATE TABLE

table definition

```
<table definition> ::= CREATE [ { <table scope> | <table type> } ] TABLE <table
name> <table contents source> [ ON COMMIT { PRESERVE | DELETE } ROWS ]

<table scope> ::= { GLOBAL | LOCAL } TEMPORARY

<table type> :: = MEMORY | CACHED

<table contents source> ::= <table element list> | <as subquery clause>

<table element list> ::= <left paren> <table element> [ { <comma> <table ele-
ment> }... ] <right paren>

<table element> ::= <column definition> | <table constraint definition> | <like
clause>
```

like clause

A `<like clause>` copies all column definitions from another table into the newly created table. Its three options indicate if the `<default clause>`, `<identity column specification>` and `<generation clause>` associated with the column definitions are copied or not. If an option is not specified, it defaults to EXCLUDING. The `<generation clause>` refers to columns that are generated by an expression but not to identity columns. All NOT NULL constraints are copied with the original columns, other constraints are not. The `<like clause>` can be used multiple times, allowing the new table to have copies of the column definitions of one or more other tables.

```
CREATE TABLE t (id INTEGER PRIMARY KEY, LIKE atable INCLUDING DEFAULTS EXCLUDING IDENTITY)
```

```
<like clause> ::= LIKE <table name> [ <like options> ]

<like options> ::= <like option>...

<like option> ::= <identity option> | <column default option> | <generation
option>

<identity option> ::= INCLUDING IDENTITY | EXCLUDING IDENTITY
```

```
<column default option> ::= INCLUDING DEFAULTS | EXCLUDING DEFAULTS
```

```
<generation option> ::= INCLUDING GENERATED | EXCLUDING GENERATED
```

as subquery clause

```
<as subquery clause> ::= [ <left paren> <column name list> <right paren> ] AS
<table subquery> { WITH NO DATA | WITH DATA }
```

An `<as subquery clause>` used in table definition creates a table based on a `<table subquery>`. This kind of table definition is similar to a view definition. If `WITH DATA` is specified, then the new table will contain the rows of data returned by the `<table subquery>`.

```
CREATE TABLE t (a, b, c) AS (SELECT * FROM atable) WITH DATA
```

column definition

A column definition consists of a `<column name>` and in most cases a `<data type>` or `<domain name>` as minimum. The other elements of `<column definition>` are optional. Each `<column name>` in a table is unique.

```
<column definition> ::= <column name> [ <data type or domain name> ] [ <default
clause> | <identity column specification> | <generation clause> ] [ <column
constraint definition>... ] [ <collate clause> ]
```

```
<data type or domain name> ::= <data type> | <domain name>
```

```
<column constraint definition> ::= [ <constraint name definition> ] <column
constraint> [ <constraint characteristics> ]
```

```
<column constraint> ::= NOT NULL | <unique specification> | <references speci-
fication> | <check constraint definition>
```

```
<identity column specification> ::= GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ <left paren> <common sequence generator options> <right paren> ]
```

```
<generation clause> ::= GENERATED ALWAYS AS <generation expression>
```

```
<generation expression> ::= <left paren> <value expression> <right paren>
```

The `<identity column specification>` can be specified for only a single column of the table.

A `<column constraint definition>` is a shortcut for a `<table constraint definition>`. A constraint that is defined in this way is automatically turned into a table constraint. A name is automatically generated for the constraint and assigned to it.

The `<identity column specification>` is used for special columns which represent values based on un-named sequence generators. It is possible to insert a row into the able without specifying a value for the column. The value is then generated by the sequence generators according to its rules. An identity column may or may not be the primary key. Example below:

```
CREATE TABLE t (id INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 100), name VARCHAR(20) PRIMARY
 KEY, )
```

The `<generation clause>` is used for special columns which represent values based on the values held in other columns in the same row. The `<value expression>` must reference only other, non-generated, columns of the table in the same row. Therefore, any function used in the expression may not access SQL-data, and no `<query`

expression> is allowed. When <generation clause> is used, <data type> or <domain name> may be omitted.

A generated column can be part of a foreign key or unique constraints or a column of an index. This capability is the main reason for using generated columns. A generated column may contain a formula that computes a value based on the values of other columns. Fast searches of the computed value can be performed when an index is declared on the generated column. Or the computed values can be declared to be unique, using a UNIQUE constraint on the table.

When a row is inserted into a table, or an existing row is updated, no value except DEFAULT can be specified for a generated column. In the example below, data is inserted into the non-generated columns and the generated column will contain 'Felix the Cat' or 'Pink Panther'.

```
CREATE TABLE t (id INTEGER PRIMARY KEY,
    firstname VARCHAR(20),
    lastname VARCHAR(20),
    fullname VARCHAR(40) GENERATED ALWAYS AS (firstname || ' ' || lastname))
INSERT INTO t (id, firstname, lastname) VALUES (1, 'Felix', 'the Cat')
INSERT INTO t (id, firstname, lastname, fullname) VALUES (2, 'Pink', 'Panther', DEFAULT)
```

DEFAULT

default clause

A default clause can be used if GENERATED is not specified. If a column has a <default clause> then it is possible to insert a row into the table without specifying a value for the column.

```
<default clause> ::= DEFAULT <default option>
```

```
<default option> ::= <literal> | <datetime value function> | USER | CURRENT_USER
| CURRENT_ROLE | SESSION_USER | SYSTEM_USER | CURRENT_CATALOG | CURRENT_SCHEMA
| CURRENT_PATH | NULL
```

The type of the <default option> must match the type of the column.

CONSTRAINT

constraint name and characteristics

```
<constraint name definition> ::= CONSTRAINT <constraint name>
```

```
<constraint characteristics> ::= <constraint check time> [ [ NOT ] DEFERRABLE
[ <constraint check time> ] ]
```

```
<constraint check time> ::= INITIALLY DEFERRED | INITIALLY IMMEDIATE
```

Specify the name of a constraint and its characteristics. By default the constraint is NOT DEFERRABLE and INITIALLY IMMEDIATE. This means the constraint is enforced as soon as a data change statement is executed. If INITIALLY DEFERRED is specified, then the constraint is enforced when the session commits. The characteristics must be compatible. The constraint check time can be changed temporarily for an SQL session. HyperSQL does not support deferring constraint enforcement. This feature of the SQL Standard has been criticised because it allows a session to read uncommitted data that violates database integrity constraints but has not yet been checked.

CONSTRAINT

table constraint definition

```
<table constraint definition> ::= [ <constraint name definition> ] <table con-
straint> [ <constraint characteristics> ]
```

```
<table constraint> ::= <unique constraint definition> | <referential constraint
definition> | <check constraint definition>
```

Three kinds of constraint can be defined on a table: UNIQUE (including PRIMARY KEY), FOREIGN KEY and CHECK. Each kind has its own rules to limit the values that can be specified for different columns in each row of the table.

UNIQUE

unique constraint definition

```
<unique constraint definition> ::= <unique specification> <left paren> <unique
column list> <right paren> | UNIQUE ( VALUE )
```

```
<unique specification> ::= UNIQUE | PRIMARY KEY
```

```
<unique column list> ::= <column name list>
```

A unique constraint is specified on a single column or on multiple columns. On each set of columns taken together, only one UNIQUE constraint can be specified. Each column of a PRIMARY KEY constraint has an implicit NOT NULL constraint.

If UNIQUE(VALUE) is specified, the constraint created on all columns of the table.

FOREIGN KEY

referential constraint definition

```
<referential constraint definition> ::= FOREIGN KEY <left paren> <referencing
columns> <right paren> <references specification>
```

```
<references specification> ::= REFERENCES <referenced table and columns> [ MATCH
<match type> ] [ <referential triggered action> ]
```

```
<match type> ::= FULL | PARTIAL | SIMPLE
```

```
<referencing columns> ::= <reference column list>
```

```
<referenced table and columns> ::= <table name> [ <left paren> <reference column
list> <right paren> ]
```

```
<reference column list> ::= <column name list>
```

```
<referential triggered action> ::= <update rule> [ <delete rule> ] | <delete
rule> [ <update rule> ]
```

```
<update rule> ::= ON UPDATE <referential action>
```

```
<delete rule> ::= ON DELETE <referential action>
```

```
<referential action> ::= CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION
```

A referential constraint allows links to be established between the rows of two tables. The specified list of `<refer-encing columns>` corresponds one by one to the columns of the specified list of `<referenced columns>` in another table (or sometimes in the same table). For each row in the table, a row must exist in the referenced table with equivalent values in the two column lists. There must exist a single unique constraint in the referenced table on all the `<referenced columns>`.

The [MATCH match type] clause is optional and has an effect only on multi-column foreign keys and only on rows containing at least a NULL in one of the <referencing columns>. If the clause is not specified, MATCH SIMPLE is the default. If MATCH SIMPLE is specified, then any NULL means the row can exist (without a corresponding row in the referenced table). If MATCH FULL is specified then either all the column values must be NULL or none of them. MATCH PARTIAL allows any NULL but the non NULL values must match those of a row in the referenced table. HyperSQL does not support MATCH PARTIAL.

Referential actions are specified with ON UPDATE and ON DELETE clauses. These actions take place when a row in the referenced table (the parent table) has referencing rows in the referencing table and it is deleted or modified with any SQL statement. The default is NO ACTION. This means the SQL statement that causes the DELETE or UPDATE is terminated with an exception. The RESTRICT option is similar and works exactly the same without deferrable constraints (which are not allowed by HyperSQL). The other three options, CASCADE, SET NULL and SET DEFAULT all allow the DELETE or UPDATE statement to complete. With DELETE statements the CASCADE option results in the referencing rows to be deleted. With UPDATE statements, the changes to the values of the referenced columns are copied to the referencing rows. With both DELETE or UPDATE statement, the SET NULL option results in the columns of the referencing rows to be set to NULL. Similarly, the SET DEFAULT option results in the columns of the referencing rows to be set to their default values.

CHECK

check constraint definition

```
<check constraint definition> ::= CHECK <left paren> <search condition> <right paren>
```

A CHECK constraint can exist for a TABLE or for a DOMAIN. The <search condition> evaluates to an SQL BOOLEAN value for each row of the table. Within the <search condition> all columns of the table row can be referenced. For all rows of the table, the <search condition> evaluates to TRUE or UNKNOWN. When a new row is inserted, or an existing row is updated, the <search condition> is evaluated and if it is FALSE, the insert or update fails.

A CHECK constraint for a DOMAIN is similar. In its <search condition>, the term VALUE is used to represents the value to which the DOMAIN applies.

```
CREATE TABLE t (a VARCHAR(20) CHECK (a IS NOT NULL AND CHARACTER_LENGTH(a) > 2))
```

The search condition of a CHECK constraint cannot contain any function that is not deterministic. A check constraint is a data integrity constraint, therefore it must hold with respect to the rest of the data in the database. It cannot use values that are temporal or ephemeral. For example CURRENT_USER is a function that returns different values depending on who is using the database, or CURRENT_DATE changes day-to-day. Some temporal expressions are retrospectively deterministic and are allowed in check constraints. For example, (CHECK VALUE < CURRENT_DATE) is valid, because CURRENT_DATE will not move backwards in time, but (CHECK VALUE > CURRENT_DATE) is not acceptable.

If you want to enforce the condition that a date value that is inserted into the database belongs to the future (at the time of insertion), or any similar constraint, then use a TRIGGER with the desired condition.

SET TABLE writeability

set table write property (HyperSQL)

```
<set table read only statement> ::= SET TABLE <table name> { READ ONLY | READ WRITE }
```

Set the writeability property of a table. Tables are writable by default. This statement can be used to change the property between READ ONLY and READ WRITE. This is a feature of HyperSQL.

SET TABLE SOURCE

set table source statement

```
<set table source statement> ::= SET TABLE <table name> SOURCE <file and options>
[DESC]

<file and options>::= <doublequote> <file path> [<semicolon> <property>...]
<doublequote>
```

Set the text source for a text table. This statement cannot be used for tables that are not defined as TEXT TABLE.

Supported Properties

quoted = { true | false }	default is true. If false, treats double quotes as normal characters
all_quoted = { true | false }	default is false. If true, adds double quotes around all fields.
encoding = <encoding name>	character encoding for text and character fields, for example, encoding=UTF-8
ignore_first = { true | false }	default is false. If true ignores the first line of the file
cache_scale= <numeric value>	exponent to calculate rows of the text file in cache. Default is 8, equivalent to nearly 800 rows
cache_size_scale = <numeric value>r	exponent to calculate average size of each row in cache. Default is 8, equivalent to 256 bytes per row.
fs = <unquoted character>	field separator
vs = <unquoted character>	varchar separator

Special indicators for HyperSQL Text Table separators

\semi	semicolon
\quote	quote
\space	space character
\apos	apostrophe
\n	newline - Used as an end anchor (like $ in regular expressions)
\r	carriage return
\t	tab
\\	backslash
\u####	a Unicode character specified in hexadecimal

In the example below, the text source of the table is set to "myfile", the field separator to the pipe symbol, and the long varchar separator to the tilde symbol.

```
    SET TABLE mytable SOURCE 'myfile;fs=|;vs=.;lvs=~'
```

Only a user with the DBA role can execute this statement.

SET TABLE SOURCE HEADER

set table source header statement

```
<set table source header statement> ::= SET TABLE <table name> SOURCE HEADER
<header string>
```

Set the header for the text source for a text table. If this command is used, the <header string> is used as the first line of the source file of the text table. This line is not part of the table data. Only a user with the DBA role can execute this statement.

SET TABLE SOURCE on-off

set table source on-off statement

```
<set table source on-off statement> ::= SET TABLE <table name> SOURCE { ON | OFF }
```

Attach or detach a text table from its text source. This command does not change the properties or the name of the file that is the source of a text table. When OFF is specified, the command detaches the table from its source and closes the file for the source. In this state, it is not possible to read or write to the table. This allows the user to replace the file with a different file, or delete it. When ON is specified, the source file is read. Only a user with the DBA role can execute this statement

ALTER TABLE

alter table statement

```
<alter table statement> ::= ALTER TABLE <table name> <alter table action>
```

```
<alter table action> ::= <add column definition> | <alter column definition>
| <drop column definition> | <add table constraint definition> | <drop table
constraint definition>
```

Change the definition of a table. Specific types of this statement are covered below.

ADD COLUMN

add column definition

```
<add column definition> ::= ADD [ COLUMN ] <column definition> [ BEFORE <other
column name> ]
```

Add a column to an existing table. The <column definition> is specified the same way as it is used in <table definition>. HyperSQL allows the use of [BEFORE <other column name>] to specify at which position the new column is added to the table.

If the table contains rows, the new column must have a <default clause> or use one of the forms of GENERAT-ED. The column values for each row is then filled with the result of the <default clause> or the generated value.

ALTER COLUMN

alter column definition

```
<alter column definition> ::= ALTER [ COLUMN ] <column name> <alter column
action>
```

```
<alter column action> ::= <set column default clause> | <drop column default
clause> | <alter column data type clause> | <alter identity column specification>
| <alter column nullability> | <alter column name>
```

Change a column and its definition. Specific types of this statement are covered below. See also the RENAME statement above.

SET DEFAULT

set column default clause

```
<set column default clause> ::= SET <default clause>
```

Set the default clause for a column. This can be used if the column is not defined as GENERATED.

DROP DEFAULT

drop column default clause

```
<drop column default clause> ::= DROP DEFAULT
```

Drop the default clause from a column.

SET DATA TYPE

alter column data type clause

```
<alter column data type clause> ::= SET DATA TYPE <data type>
```

Change the declared type of a column. The (proposed) SQL Standard allows only changes to type properties such as maximum length, precision, or scale, and only changes that cause the property to enlarge. HyperSQL allows changing the type if all the existing values can be cast into the new type without string truncation or loss of significant digits.

alter identity column

alter identity column specification

```
<alter identity column specification> ::= <alter identity column option>...

<alter identity column option> ::= <alter sequence generator restart option> |
SET <basic sequence generator option>
```

Change the properties of an identity column. This command is similar to the commands used for changing the properties of named SEQUENCE objects discussed in this section.

SET NULL

alter column nullability

```
<alter column nullability> ::= SET NULL
```

Removes a NOT NULL constraint from a column. This option is specific to HyperSQL

DROP COLUMN

drop column definition

```
<drop column definition> ::= DROP [ COLUMN ] <column name> <drop behavior>
```

Destroy a column of a base table. The `<drop behavior>` is either `RESTRICT` or `CASCADE`. If the column is referenced in a table constraint that references other columns as well as this column, or if the column is referenced in a VIEW, or the column is referenced in a TRIGGER, then the statement will fail if `RESTRICT` is specified. If `CASCADE` is specified, then any CONSTRAINT, VIEW or TRIGGER object that references the column is dropped with a cascading effect.

ADD CONSTRAINT

add table constraint definition

```
<add table constraint definition> ::= ADD <table constraint definition>
```

Add a constraint to a table. The existing rows of the table must conform to the added constraint, otherwise the statement will not succeed.

DROP CONSTRAINT

drop table constraint definition

```
<drop table constraint definition> ::= DROP CONSTRAINT <constraint name> <drop
behavior>
```

Destroy a constraint on a table. The `<drop behavior>` has an effect only on UNIQUE and PRIMARY KEY constraints. If such a constraint is referenced by a FOREIGN KEY constraint, the FOREIGN KEY constraint will be dropped if `CASCADE` is specified. If the columns of such a constraint are used in a GROUP BY clause in the query expression of a VIEW or another kind of schema object, and a functional dependency relationship exists between these columns and the other columns in that query expression, then the VIEW or other schema object will be dropped when `CASCADE` is specified.

DROP TABLE

drop table statement

```
<drop table statement> ::= DROP TABLE [ IF EXISTS ] <table name> [ IF EXISTS ]
<drop behavior>
```

Destroy a table. The default drop behaviour is RESTRICT and will cause the statement to fail if there is any view or foreign key constraint that references the table. If `<drop behavior>` is `CASCADE`, it causes all schema objects that reference the table to drop. Referencing views are dropped. In the case of foreign key constraints that reference the table, the constraint is dropped, rather than the TABLE or DOMAIN that contains it.

View Creation and Manipulation

CREATE VIEW

view definition

```
<view definition> ::= CREATE [ RECURSIVE ] VIEW <table name> <view specification>
AS <query expression> [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]

<view specification> ::= [ <left paren> <view column list> <right paren> ]

<view column list> ::= <column name list>
```

Define a view. The `<query expression>` is a SELECT or similar statement. The `<view column list>` is the list of unique names for the columns of the view. The number of columns in the `<view column list>`

must match the number of columns returned by the `<query expression>`. If `<view column list>` is not specified, then the columns of the `<query expression>` should have unique names and are used as the names of the view column.

Some views are updatable. As covered elsewhere, an updatable view is based on a single table or updatable view. For updatable views, the optional `CHECK OPTION` clause can be specified. If this option is specified, then if a row of the view is updated or a new row is inserted into the view, then it should contain such values that the row would be included in the view after the change. If `WITH CASCADED CHECK OPTION` is specified, then if the `<query expression>` of the view references another view, then the search condition of the underlying view should also be satisfied by the update or insert operation.

More on recursive...

DROP VIEW

drop view statement

```
<drop view statement> ::= DROP VIEW [ IF EXISTS ] <table name> [ IF EXISTS ]
<drop behavior>
```

Destroy a view. The `<drop behavior>` is similar to dropping a table.

ALTER VIEW

alter view statement

```
<alter view statement> ::= ALTER VIEW <table name> <view specification> AS
<query expression> [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Alter a view. The statement is otherwise identical to CREATE VIEW. The new definition replaces the old. If there are database objects such as routines or views that reference the view, then these objects are recompiled with the new view definition. If the new definition is not compatible, the statement fails.

Domain Creation and Manipulation

CREATE DOMAIN

domain definition

```
<domain definition> ::= CREATE DOMAIN <domain name> [ AS ] <predefined type>
[ <default clause> ] [ <domain constraint>... ] [ <collate clause> ]

<domain constraint> ::= [ <constraint name definition> ] <check constraint
definition> [ <constraint characteristics> ]
```

Define a domain. Although a DOMAIN is not strictly a type in the SQL Standard, it can be informally considered as a type. A DOMAIN is based on a `<predefined type>`, which is a base type defined by the Standard. It can have a `<default clause>`, similar to a column default clause. It can also have one or more CHECK constraints which limit the values that can be assigned to a column or variable that has the DOMAIN as its type.

```
CREATE DOMAIN valid_string AS VARCHAR(20) DEFAULT 'NO VALUE' CHECK (value IS NOT NULL AND
 CHARACTER_LENGTH(value) > 2)
```

ALTER DOMAIN

alter domain statement

```
<alter domain statement> ::= ALTER DOMAIN <domain name> <alter domain action>
```

```
<alter domain action> ::= <set domain default clause> | <drop domain default
clause> | <add domain constraint definition> | <drop domain constraint defi-
nition>
```

Change a domain and its definition.

SET DEFAULT

set domain default clause

```
<set domain default clause> ::= SET <default clause>
```

Set the default value in a domain.

DROP DEFAULT

drop domain default clause

```
<drop domain default clause> ::= DROP DEFAULT
```

Remove the default clause of a domain.

ADD CONSTRAINT

add domain constraint definition

```
<add domain constraint definition> ::= ADD <domain constraint>
```

Add a constraint to a domain.

DROP CONSTRAINT

drop domain constraint definition

```
<drop domain constraint definition> ::= DROP CONSTRAINT <constraint name>
```

Destroy a constraint on a domain. If the `<drop behavior>` is CASCADE, and the constraint is a UNIQUE constraint which is referenced by a FOREIGN KEY constraint on another table, then the FOREIGN KEY constraint is also dropped.

DROP DOMAIN

drop domain statement

```
<drop domain statement> ::= DROP DOMAIN <domain name> <drop behavior>
```

Destroy a domain. If `<drop behavior>` is CASCADE, it works differently from most other objects. If a table features a column of the specified DOMAIN, the column survives and inherits the DEFAULT CLAUSE, and the CHECK CONSTRAINT of the DOMAIN.

Trigger Creation

CREATE TRIGGER

trigger definition

```
<trigger definition> ::= CREATE TRIGGER <trigger name> <trigger action time>
<trigger event> ON <table name> [ REFERENCING <transition table or variable
list> ] <triggered action>

<trigger action time> ::= BEFORE | AFTER | INSTEAD OF

<trigger event> ::= INSERT | DELETE | UPDATE [ OF <trigger column list> ]

<trigger column list> ::= <column name list>

<triggered action> ::= [ FOR EACH { ROW | STATEMENT } ] [ <triggered when
clause> ] <triggered SQL statement>

<triggered when clause> ::= WHEN <left paren> <search condition> <right paren>

<triggered SQL statement> ::= <SQL procedure statement> | BEGIN ATOMIC { <SQL
procedure statement> <semicolon> }... END | [QUEUE <integer literal>] [NOWAIT]
CALL <HSQLDB trigger class FQN>

<transition table or variable list> ::= <transition table or variable>...

<transition table or variable> ::= OLD [ ROW ] [ AS ] <old transition variable
name> | NEW [ ROW ] [ AS ] <new transition variable name> | OLD TABLE [ AS ]
<old transition table name> | NEW TABLE [ AS ] <new transition table name>

<old transition table name> ::= <transition table name>

<new transition table name> ::= <transition table name>

<transition table name> ::= <identifier>

<old transition variable name> ::= <correlation name>

<new transition variable name> ::= <correlation name>
```

Trigger definition is a relatively complex statement. The combination of `<trigger action time>` and `<trigger event>` determines the type of the trigger. Examples include BEFORE DELETE, AFTER UPDATE, INSTEAD OF INSERT. If the optional `[ OF <trigger column list> ]` is specified for an UPDATE trigger, then the trigger is activated only if one of the columns that is in the `<trigger column list>` is specified in the UPDATE statement that activates the trigger.

If a trigger is `FOR EACH ROW`, which is the default option, then the trigger is activated for each row of the table that is affected by the execution of an SQL statement. Otherwise, it is activated once only per statement execution. In the first case, there is a before and after state for each row. For UPDATE triggers, both before and after states exist, representing the row before the update, and after the update. For DELETE, triggers, there is only a before state. For INSERT triggers, there is only an after state. If a trigger is `FOR EACH STATEMENT`, then a transient table is created containing all the rows for the before state and another transient table is created for the after state.

The `[ REFERENCING <transition table or variable> ]` is used to give a name to the before and after data row or table. This name can be referenced in the `<SQL procedure statement>` to access the data.

The optional `<triggered when clause>` is a search condition, similar to the search condition of a DELETE or UPDATE statement. If the search condition is not TRUE for a row, then the trigger is not activated for that row.

The `<SQL procedure statement>` is limited to INSERT, DELETE, UPDATE and MERGE statements.

The `<HSQLDB trigger class FQN>` is a delimited identifier that contains the fully qualified name of a Java class that implements the `org.hsqldb.Trigger` interface.

Early releases of HyperSQL version 2.0 do not allow the use of OLD TABLE or NEW TABLE in statement level trigger definitions.

DROP TRIGGER

drop trigger statement

```
<drop trigger statement> ::= DROP TRIGGER <trigger name>
```

Destroy a trigger.

Routine Creation

schema routine

SQL-invoked routine

```
<SQL-invoked routine> ::= <schema routine>

<schema routine> ::= <schema procedure> | <schema function>

<schema procedure> ::= CREATE <SQL-invoked procedure>

<schema function> ::= CREATE <SQL-invoked function>

<SQL-invoked procedure> ::= PROCEDURE <schema qualified routine name> <SQL
parameter declaration list> <routine characteristics> <routine body>

<SQL-invoked function> ::= { <function specification> | <method specification
designator> } <routine body>

<SQL parameter declaration list> ::= <left paren> [ <SQL parameter declaration>
[ { <comma> <SQL parameter declaration> }... ] ] <right paren>

<SQL parameter declaration> ::= [ <parameter mode> ] [ <SQL parameter name> ]
<parameter type> [ RESULT ]

<parameter mode> ::= IN | OUT | INOUT

<parameter type> ::= <data type>

<function specification> ::= FUNCTION <schema qualified routine name> <SQL pa-
rameter declaration list> <returns clause> <routine characteristics> [ <dispatch
clause> ]

<method specification designator> ::= SPECIFIC METHOD <specific method name>
| [ INSTANCE | STATIC | CONSTRUCTOR ] METHOD <method name> <SQL parameter
declaration list> [ <returns clause> ] FOR <schema-resolved user-defined type
name>

<routine characteristics> ::= [ <routine characteristic>... ]

<routine characteristic> ::= <language clause> | <parameter style clause> |
SPECIFIC <specific name> | <deterministic characteristic> | <SQL-data access
indication> | <null-call clause> | <returned result sets characteristic> |
<savepoint level indication>
```

```
<savepoint level indication> ::= NEW SAVEPOINT LEVEL | OLD SAVEPOINT LEVEL

<returned result sets characteristic> ::= DYNAMIC RESULT SETS <maximum returned
result sets>

<parameter style clause> ::= PARAMETER STYLE <parameter style>

<dispatch clause> ::= STATIC DISPATCH

<returns clause> ::= RETURNS <returns type>

<returns type> ::= <returns data type> [ <result cast> ] | <returns table type>

<returns table type> ::= TABLE <table function column list>

<table function column list> ::= <left paren> <table function column list
element> [ { <comma> <table function column list element> }... ] <right paren>

<table function column list element> ::= <column name> <data type>

<result cast> ::= CAST FROM <result cast from type>

<result cast from type> ::= <data type> [ <locator indication> ]

<returns data type> ::= <data type> [ <locator indication> ]

<routine body> ::= <SQL routine spec> | <external body reference>

<SQL routine spec> ::= [ <rights clause> ] <SQL routine body>

<rights clause> ::= SQL SECURITY INVOKER | SQL SECURITY DEFINER

<SQL routine body> ::= <SQL procedure statement>

<external body reference> ::= EXTERNAL [ NAME <external routine name> ] [ <pa-
rameter style clause> ]

<parameter style> ::= SQL | GENERAL

<deterministic characteristic> ::= DETERMINISTIC | NOT DETERMINISTIC

<SQL-data access indication> ::= NO SQL | CONTAINS SQL | READS SQL DATA |
MODIFIES SQL DATA

<null-call clause> ::= RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

<maximum returned result sets> ::= <unsigned integer>
```

Define an SQL-invoked routine.

ALTER routine

alter routine statement

```
<alter routine statement> ::= ALTER <specific routine designator> <alter routine
characteristics> <alter routine behavior>

<alter routine characteristics> ::= <alter routine characteristic>...
```

```
<alter routine characteristic> ::= <language clause> | <parameter style clause>
| <SQL-data access indication> | <null-call clause> | <returned result sets
characteristic> | NAME <external routine name>
```

```
<alter routine behavior> ::= RESTRICT
```

Alter a characteristic of an SQL-invoked routine. Early releases of HyperSQL 2.0 may not support this statement.

DROP

drop routine statement

```
<drop routine statement> ::= DROP <specific routine designator> <drop behavior>
```

Destroy an SQL-invoked routine.

Sequence Creation

CREATE SEQUENCE

sequence generator definition

```
<sequence generator definition> ::= CREATE SEQUENCE <sequence generator name>
[ <sequence generator options> ]
```

```
<sequence generator options> ::= <sequence generator option> ...
```

```
<sequence generator option> ::= <sequence generator data type option> | <common
sequence generator options>
```

```
<common sequence generator options> ::= <common sequence generator option> ...
```

```
<common sequence generator option> ::= <sequence generator start with option>
| <basic sequence generator option>
```

```
<basic sequence generator option> ::= <sequence generator increment by option>
| <sequence generator maxvalue option> | <sequence generator minvalue option>
| <sequence generator cycle option>
```

```
<sequence generator data type option> ::= AS <data type>
```

```
<sequence generator start with option> ::= START WITH <sequence generator start
value>
```

```
<sequence generator start value> ::= <signed numeric literal>
```

```
<sequence generator increment by option> ::= INCREMENT BY <sequence generator
increment>
```

```
<sequence generator increment> ::= <signed numeric literal>
```

```
<sequence generator maxvalue option> ::= MAXVALUE <sequence generator max value>
| NO MAXVALUE
```

```
<sequence generator max value> ::= <signed numeric literal>
```

```
<sequence generator minvalue option> ::= MINVALUE <sequence generator min value>
| NO MINVALUE
```

```
<sequence generator min value> ::= <signed numeric literal>

<sequence generator cycle option> ::= CYCLE | NO CYCLE
```

Define a named sequence generator. A SEQUENCE object generates a sequence of integers according to the specified rules. The simple definition without the options defines a sequence of numbers in INTEGER type starting at 1 and incrementing by 1. By default the CYCLE property is set and the minimum and maximum limits are the minimum and maximum limits of the type of returned values. There are self-explanatory options for changing various properties of the sequence. The MAXVALUE and MINVALUE specify the upper and lower limits. If CYCLE is specified, after the sequence returns the highest or lowest value in range, the next value will respectively be the lowest or highest value in range. If NO CYCLE is specified, the use of the sequence generator results in an error once the limit has been reached.

The integer types: SMALLINT, INTEGER, BIGINT, DECIMAL and NUMERIC can be used as the type of the sequence. DECIMAL and NUMERIC types must have a scale of 0 and a precision not exceeding 18.

ALTER SEQUENCE

alter sequence generator statement

```
<alter sequence generator statement> ::= ALTER SEQUENCE <sequence generator
name> <alter sequence generator options>

<alter sequence generator options> ::= <alter sequence generator option>...

<alter sequence generator option> ::= <alter sequence generator restart option>
| <basic sequence generator option>

<alter sequence generator restart option> ::= RESTART [ WITH <sequence generator
restart value> ]

<sequence generator restart value> ::= <signed numeric literal>
```

Change the definition of a named sequence generator. The same options that are used in the definition of the SEQUENCE can be used to alter it. The exception is the option for the start value which is RESTART WITH for the ALTER SEQUENCE statement..

DROP SEQUENCE

drop sequence generator statement

```
<drop sequence generator statement> ::= DROP SEQUENCE [ IF EXISTS ] <sequence
generator name> [ IF EXISTS ] <drop behavior>
```

Destroy an external sequence generator. If the <drop behavior> is CASCADE, then all objects that reference the sequence are dropped. These objects can be VIEW, ROUTINE or TRIGGER objects.

SQL Procedure Statement

SQL procedure statement

SQL procedure statement

The definition of CREATE TRIGGER and CREATE PROCEDURE statements refers to <SQL procedure statement>. The definition of this element is given below. However, only a subset of these statements are allowed in trigger or routine definition.

```
<SQL procedure statement> ::= <SQL executable statement>
```

```
<SQL executable statement> ::= <SQL schema statement> | <SQL data statement>
| <SQL control statement> | <SQL transaction statement> | <SQL connection
statement> | <SQL session statement> | <SQL diagnostics statement> | <SQL dynamic
statement>

<SQL schema statement> ::= <SQL schema definition statement> | <SQL schema
manipulation statement>

<SQL schema definition statement> ::= <schema definition> | <table definition> |
<view definition> | <SQL-invoked routine> | <grant statement> | <role definition>
| <domain definition> | <character set definition> | <collation definition> |
<transliteration definition> | <assertion definition> | <trigger definition> |
<user-defined type definition> | <user-defined cast definition> | <user-defined
ordering definition> | <transform definition> | <sequence generator definition>

<SQL schema manipulation statement> ::= <drop schema statement> | <alter table
statement> | <drop table statement> | <drop view statement> | <alter routine
statement> | <drop routine statement> | <drop user-defined cast statement> |
<revoke statement> | <drop role statement> | <alter domain statement> | <drop
domain statement> | <drop character set statement> | <drop collation statement>
| <drop transliteration statement> | <drop assertion statement> | <drop trigger
statement> | <alter type statement> | <drop data type statement> | <alter
sequence generator statement> | <drop sequence generator statement>
```

Other Schema Object Creation

CREATE INDEX

create index statement

```
<create index statement> ::= CREATE INDEX <index name> ON <table name> <left
paren> {<column name> [ASC | DESC]}, ... <left paren>
```

Creates an index on a group of columns of a table. The optional [ASC | DESC] specifies if the column is indexed in the ascending or descending order, but has no effect on how the index is created (it is allowed for compatibility with other database engines). HyperSQL can use all indexes in ascending or descending order as needed.

DROP INDEX

drop index statement

```
<drop index statement> ::= DROP INDEX [ IF EXISTS ] <index name> [ IF EXISTS ]
```

Destroy an index.

CREATE TYPE

user-defined type definition

```
<user-defined type definition> ::= CREATE TYPE <user-defined type body>

<user-defined type body> ::= <schema-resolved user-defined type name> [ AS
<representation> ]

<representation> ::= <predefined type>
```

Define a user-defined type. Currently only simple distinct types can be defined without further attributes.

CREATE CAST

user-defined cast definition

```
<user-defined cast definition> ::= CREATE CAST <left paren> <source data type>
AS <target data type> <right paren> WITH <cast function> [ AS ASSIGNMENT ]

<cast function> ::= <specific routine designator>

<source data type> ::= <data type>

<target data type> ::= <data type>
```

Define a user-defined cast. This feature may be supported in a future versions of HyperSQL.

DROP CAST

drop user-defined cast statement

```
<drop user-defined cast statement> ::= DROP CAST <left paren> <source data type>
AS <target data type> <right paren> <drop behavior>
```

Destroy a user-defined cast. This feature may be supported in a future versions of HyperSQL.

CREATE CHARACTER SET

character set definition

```
<character set definition> ::= CREATE CHARACTER SET <character set name> [ AS ]
<character set source> [ <collate clause> ]

<character set source> ::= GET <character set specification>
```

Define a character set. A new CHARACTER SET is based on an existing CHARACTER SET. The optional `<col-late clause>` specifies the collation to be used, otherwise the collation is inherited from the default collation for the source CHARACTER SET.

DROP CHARACTER SET

drop character set statement

```
<drop character set statement> ::= DROP CHARACTER SET <character set name>
```

Destroy a character set. If the character set name is referenced in any database object, the command fails. Note that `CASCADE` or `RESTRICT` cannot be specified for this command.

CREATE COLLATION

collation definition

```
<collation definition> ::= CREATE COLLATION <collation name> FOR <character set
specification> FROM <existing collation name> [ <pad characteristic> ]

<existing collation name> ::= <collation name>

<pad characteristic> ::= NO PAD | PAD SPACE
```

Define a collation. A new collation is based on an existing COLLATION and applies to an existing CHARACTER SET. The `<pad characteristic>` specifies whether strings are padded with spaces for comparison. This feature may be supported in a future versions of HyperSQL.

DROP COLLATION

drop collation statement

```
<drop collation statement> ::= DROP COLLATION <collation name> <drop behavior>
```

Destroy a collation. If the `<drop behavior>` is CASCADE, then all references to the collation revert to the default collation that would be in force if the dropped collation was not specified. This feature may be supported in a future versions of HyperSQL.

CREATE TRANSLATION

transliteration definition

```
<transliteration definition> ::= CREATE TRANSLATION <transliteration name> FOR
<source character set specification> TO <target character set specification>
FROM <transliteration source>

<source character set specification> ::= <character set specification>

<target character set specification> ::= <character set specification>

<transliteration source> ::= <existing transliteration name> | <transliteration
routine>

<existing transliteration name> ::= <transliteration name>

<transliteration routine> ::= <specific routine designator>
```

Define a character transliteration. This feature may be supported in a future versions of HyperSQL.

DROP TRANSLATION

drop transliteration statement

```
<drop transliteration statement> ::= DROP TRANSLATION <transliteration name>
```

Destroy a character transliteration. This feature may be supported in a future versions of HyperSQL.

CREATE ASSERTION

assertion definition

```
<assertion definition> ::= CREATE ASSERTION <constraint name> CHECK <left paren>
<search condition> <right paren> [ <constraint characteristics> ]
```

Specify an integrity constraint. This feature may be supported in a future versions of HyperSQL.

DROP ASSERTION

drop assertion statement

```
<drop assertion statement> ::= DROP ASSERTION <constraint name> [ <drop behav-
ior> ]
```

Destroy an assertion. This feature may be supported in a future versions of HyperSQL.

The Information Schema

The Information Schema is a special schema in each catalog. The SQL Standard defines a number of character sets and domains in this schema. In addition, all the implementation-defined collations belong to the Information Schema.

The SQL Standard defines many views in the Information Schema. These views show the properties of the database objects that currently exist in the database. When a user accesses one these views, only the properties of database objects that the user can access are included.

HyperSQL supports all the views defined by the Standard, apart from a few views that report on extended user-defined types and other optional features of the Standard that are not supported by HyperSQL.

HyperSQL also adds some views to the Information Schema. These views are for features that are not reported in any of the views defined by the Standard, or for use by JDBC DatabaseMetaData.

Predefined Character Sets, Collations and Domains

The SQL Standard defines a number of character sets and domains in the INFORMATION SCHEMA.

These domains are used in the INFORMATION SCHEMA views:

CARDINAL_NUMBER, YES_OR_NO, CHARACTER_DATA, SQL_IDENTIFIER, TIME_STAMP

All available collations are in the INFORMATION SCHEMA.

Views in INFORMATION SCHEMA

The following views are defined by the SQL Standard:

ADMINISTRABLE_ROLE_AUTHORIZATIONS

APPLICABLE_ROLES

ASSERTIONS

AUTHORIZATIONS

CHARACTER_SETS

CHECK_CONSTRAINTS

CHECK_CONSTRAINT_ROUTINE_USAGE

COLLATIONS

COLUMNS

COLUMN_COLUMN_USAGE

COLUMN_DOMAIN_USAGE

COLUMN_PRIVILEGES

COLUMN_UDT_USAGE

CONSTRAINT_COLUMN_USAGE

CONSTRAINT_TABLE_USAGE

DATA_TYPE_PRIVILEGES

DOMAINS

DOMAIN_CONSTRAINTS

ENABLED_ROLES

INFORMATION_SCHEMA_CATALOG_NAME

KEY_COLUMN_USAGE

PARAMETERS

REFERENTIAL_CONSTRAINTS

ROLE_AUTHORIZATION_DESCRIPTORS

ROLE_COLUMN_GRANTS

ROLE_ROUTINE_GRANTS

ROLE_TABLE_GRANTS

ROLE_UDT_GRANTS

ROLE_USAGE_GRANTS

ROUTINE_COLUMN_USAGE

ROUTINE_JAR_USAGE

ROUTINE_PRIVILEGES

ROUTINE_ROUTINE_USAGE

ROUTINE_SEQUENCE_USAGE

ROUTINE_TABLE_USAGE

ROUTINES

SCHEMATA

SEQUENCES

SQL_FEATURES

SQL_IMPLEMENTATION_INFO

SQL_PACKAGES

SQL_PARTS

SQL_SIZING

SQL_SIZING_PROFILES

TABLES

TABLE_CONSTRAINTS

TABLE_PRIVILEGES

TRANSLATIONS

TRIGGERED_UPDATE_COLUMNS

TRIGGERS

TRIGGER_COLUMN_USAGE

TRIGGER_ROUTINE_USAGE

TRIGGER_SEQUENCE_USAGE

TRIGGER_TABLE_USAGE

USAGE_PRIVILEGES

USER_DEFINED_TYPES

VIEWS

VIEW_COLUMN_USAGE

VIEW_ROUTINE_USAGE

VIEW_TABLE_USAGE

The following views are specific to HyperSQL:

SYSTEM_BESTROWIDENTIFIER

SYSTEM_CACHEINFO

SYSTEM_COLUMNS

SYSTEM_COMMENTS

SYSTEM_CROSSREFERENCE

SYSTEM_INDEXINFO

SYSTEM_PRIMARYKEYS

SYSTEM_PROCEDURECOLUMNS

SYSTEM_PROCEDURES

SYSTEM_PROPERTIES

SYSTEM_SCHEMAS

SYSTEM_SEQUENCES

SYSTEM_SESSIONINFO

SYSTEM_SESSIONS

SYSTEM_TABLES

SYSTEM_TABLETYPES

SYSTEM_TEXTTABLES

SYSTEM_TYPEINFO

SYSTEM_UDTS

SYSTEM_USERS

SYSTEM_VERSIONCOLUMNS

Chapter 5. Text Tables

Text Tables as a Standard Feature of Hsqldb

Bob Preston, The HSQL Development Group
Fred Toussi, The HSQL Development Group

$Revision: 3601 $

Copyright 2002-2010 Bob Preston and Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2010-05-31 20:17:47 -0400 (Mon, 31 May 2010) $

Overview

Text Table support for HSQLDB was originally developed by Bob Preston independently from the Project. Subsequently Bob joined the Project and incorporated this feature into version 1.7.0, with a number of enhancements, especially the use of conventional SQL commands for specifying the files used for Text Tables.

In a nutshell, Text Tables are CSV or other delimited files treated as SQL tables. Any ordinary CSV or other delimited file can be used. The full range of SQL queries can be performed on these files, including SELECT, INSERT, UPDATE and DELETE. Indexes and unique constraints can be set up, and foreign key constraints can be used to enforce referential integrity between Text Tables themselves or with conventional tables.

The delimited file can be created by the engine, or an existing file can be used.

HyperSQL with Text Table support is the only comprehensive solution that employs the power of SQL and the universal reach of JDBC to handle data stored in text files.

The Implementation

Definition of Tables

Text Tables are defined similarly to conventional tables with the added TEXT keyword:

```
CREATE TEXT TABLE <tablename> (<column definition> [<constraint definition>])
```

The table is at first empty and cannot be written to. An additional SET command specifies the file and the separator character that the Text table uses:

```
SET TABLE <tablename> SOURCE <quoted_filename_and_options> [DESC]
```

Text Tables cannot be created in *mem:* (all-in-memory) databases (databases that have no script file).

Scope and Reassignment

• A Text table without a file assigned to it is READ ONLY and EMPTY.

• Reassigning a Text Table definition to a new file has implications in the following areas:

  1. The user is required to be an administrator.

2. Existing transactions are committed at this point.

3. Constraints, including foreign keys referencing this table, are kept intact. It is the responsibility of the adminis-
   trator to ensure their integrity.

The new source file is scanned and indexes are built when it is assigned to the table. At this point any violation of
NOT NULL, UNIQUE or PRIMARY KEY constraints are caught and the assignment is aborted. However, foreign
key constraints are not checked at the time of assignment or reassignment of the source file.

Null Values in Columns of Text Tables

• Empty fields are treated as NULL. These are fields where there is nothing or just spaces between the separators.

• Quoted empty strings are treated as empty strings.

Configuration

The default field separator is a comma (,). A different field separator can be specified within the SET TABLE SOURCE
statement. For example, to change the field separator for the table mytable to a vertical bar, place the following in the
SET TABLE SOURCE statement, for example:

```
SET TABLE mytable SOURCE "myfile;fs=|"
```

Since HSQLDB treats CHAR and VARCHAR strings the same, the ability to assign a different separator to the latter
is provided. When a different separator is assigned to a VARCHAR, it will terminate any CSV field of that type. For
example, if the first field is CHAR, and the second field VARCHAR, and the separator fs has been defined as the pipe
(|) and vs as the period (.) then the data in the CSV file for a row will look like:

```
First field data|Second field data.Third field data
```

This facility in effect offers an extra, special separator which can be used in addition to the global separator. The
following example shows how to change the default separator to the pipe (|), VARCHAR separator to the period (.)
within a SET TABLE SOURCE statement:

```
SET TABLE mytable SOURCE "myfile;fs=|;vs=."
```

HSQLDB also recognises the following special indicators for separators:

special indicators for separators

\semi   semicolon

\quote   single-quote

\space   space character

\apos   apostrophe

\n   newline - Used as an end anchor (like $ in regular expressions)

\r   carriage return

\t   tab

\\   backslash

\u####   a Unicode character specified in hexadecimal

Furthermore, HSQLDB provides csv file support with three additional boolean options: `ignore_first`, `quoted` and `all_quoted`. The `ignore_first` option (default false) tells HSQLDB to ignore the first line in a file. This option is used when the first line of the file contains column headings. The `all_quoted` option (default false) tells the program that it should use quotes around all character fields when writing to the source file. The `quoted` option (default true) uses quotes only when necessary to distinguish a field that contains the separator character. It can be set to false to prevent the use of quoting altogether and treat quote characters as normal characters. These options may be specified within the `SET TABLE SOURCE` statement:

```
    SET TABLE mytable SOURCE "myfile;ignore_first=true;all_quoted=true"
```

When the default options `all_quoted= false` and `quoted=true` are in force, fields that are written to a line of the csv file will be quoted only if they contain the separator or the quote character. The quote character is doubled when used inside a string. When `all_quoted=false` and `quoted=false` the quote character is not doubled. With this option, it is not possible to insert any string containing the separator into the table, as it would become impossible to distinguish from a separator. While reading an existing data source file, the program treats each individual field separately. It determines that a field is quoted only if the first character is the quote character. It interprets the rest of the field on this basis.

The character encoding for the source file is `ASCII` by default. To support UNICODE or source files prepared with different encodings this can be changed to `UTF-8` or any other encoding. The default is `encoding=ASCII` and the option `encoding=UTF-8` or other supported encodings can be used.

Finally, HSQLDB provides the ability to read a text file as READ ONLY, by placing the keyword "DESC" at the end of the SET TABLE SOURCE statement:

```
    SET TABLE mytable SOURCE "myfile" DESC
```

Text table source files are cached in memory. The maximum number of rows of data that are in memory at any time is controlled by the `textdb.cache_scale` property. The default value for `textdb.cache_scale` is 10 and can be changed by setting the property in the .properties file for the database. The number of rows in memory is calculated as 3*(2**scale), which translates to 3072 rows for the default textdb.cache_scale setting (10). The property can also be set for individual text tables:

```
    SET TABLE mytable SOURCE "myfile;ignore_first=true;all_quoted=true;cache_scale=12"
```

Disconnecting Text Tables

Text tables may be *disconnected* from their underlying data source, i.e. the text file.

You can explicitly disconnect a text table from its file by issuing the following statement:

```
    SET TABLE mytable SOURCE OFF
```

Subsequently, `mytable` will be empty and read-only. However, the data source description will be preserved, and the table can be re-connected to it with

```
    SET TABLE mytable SOURCE ON
```

When a database is opened, if the source file for an existing text table is missing the table remains disconnected from its data source, but the source description is preserved. This allows the missing source file to be added to the directory and the table re-connected to it with the above command.

Disconnecting text tables from their source has several uses. While disconnected, the text source can be edited outside HSQLDB provided data integrity is respected. When large text sources are used, and several constraints or indexes need to be created on the table, it is possible to disconnect the source during the creation of constraints and indexes and reduce the time it takes to perform the operation.

Text File Usage

The following information applies to the usage of text tables.

Text File Issues

- File locations are restricted to below the directory that contains the database, unless the `textdb.allow_full_path` property is set true as a Java system property. This feature is for security, otherwise an admin database user may be able to open random files.

- Blank lines are allowed anywhere in the text file, and are ignored.

- It is possible to define a primary key, identity column, unique, foreign key and check constraints for text tables.

- When a table source file is used with the `ignore_first=true` option, the first, ignored line is replaced with a blank line after a SHUTDOWN COMPACT, unless the SOURCE HEADER statement has been used.

- An existing table source file may include CHARACTER fields that do not begin with the quote character but contain instances of the quote character. These fields are read as literal strings. Alternatively, if any field begins with the quote character, then it is interpreted as a quoted string that should end with the quote character and any instances of the quote character within the string is doubled. When any field containing the quote character or the separator is written out to the source file by the program, the field is enclosed in quote character and any instance of the quote character inside the field is doubled.

- Inserts or updates of CHARACTER type field values are allowed with strings that contains the linefeed or the carriage return character. This feature is disabled when both quoted and all_quoted properties are false.

- ALTER TABLE commands that add or drop columns or constraints (apart from check constraints) are not supported with text tables that are connected to a source. First use the SET TABLE <name> SOURCE OFF, make the changes, then turn the source ON.

Text File Global Properties

Complete list of supported global properties in *.properties files

- `textdb.fs`

- `textdb.vs`

- `textdb.quoted`

- `textdb.all_quoted`

- `textdb.ignore_first`

- `textdb.encoding`

- `textdb.cache_scale`

- `textdb.allow_full_path`

Transactions

Text tables fully support transactions. New or changed rows that have not been committed are not updated in the source file. Therefore the source file always contains committed rows.

However, text tables are not as resilient to machine crashes as other types of tables. If the crash happens while the text source is being written to, the text source may contain only some of the changes made during a committed transaction. With other types of tables, additional mechanisms ensure the integrity of the data and this situation will not arise.

Chapter 6. Access Control

Fred Toussi, The HSQL Development Group

$Revision: 3096 $

Copyright 2010 Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2009-08-09 17:50:39 +0100 (Sun, 09 Aug 2009) $

Overview

Apart from schemas and their object, each HyperSQL catalog has USER and ROLE objects. These objects are collectively called *authorizations*. Each AUTHORIZATION has some access rights on some of the schemas or the objects they contain. The persistent elements of an SQL environment are database objects

Each database object has a name. A name is an identifier and is unique within its name-space. Authorizations names follow the rules described below and the case-normal form is stored in the database. When connecting to a database, the user name and password must match the case of the case-normal form.

identifier

definition of identifier

```
<identifier> ::= <regular identifier> | <delimited identifier> | <SQL language
identifier>

<delimited identifier> ::= <double quote> <character sequence> <double quote>

<regular identifier> ::= <special character sequence>

<SQL language identifier> ::= <special character sequence>
```

A `<delimited identifier>` is a sequence of characters enclosed with double-quote symbols. All characters are allowed in the character sequence.

A `<regular identifier>` is a special sequence of characters. It consists of letters, digits and the underscore characters. It must begin with a letter.

A `<SQL language identifier>` is similar to `<regular identifier>` but the letters can range only from A-Z in the ASCII character set. This type of identifier is used for names of CHARACTER SET objects.

If the character sequence of a delimited identifier is the same as an undelimited identifier, it represents the same identifier. For example "JOHN" is the same identifier as JOHN. In a `<regular identifier>` the case-normal form is considered for comparison. This form consists of the upper-case of equivalent of all the letters.

The character sequence length of all identifiers must be between 1 and 128 characters.

A reserved word is one that is used by the SQL Standard for special purposes. It is similar to a `<regular identifier>` but it cannot be used as an identifier for user objects. If a reserved word is enclosed in double quote characters, it becomes a quoted identifier and can be used for database objects.

Authorizations and Access Control

In general, ROLE and USER objects simply control access to schema objects. This is the scope of the SQL Standard. However, there are special roles that allow the creation of USER and ROLE objects and also allow some special

operations on the database as a whole. These roles are not defined by the Standard, which has left it to implementors to define such roles as they are needed for the particular SQL implementation.

A ROLE has a name a collection of zero or more other roles, plus some privileges (access rights). A USER has a name and a password. It similarly has a collection of zero or more roles plus some privileges.

USER objects existed in the SQL-92, but ROLE objects were introduced in SQL:1999. Originally it was intended that USER objects would normally be the same as the operating system USER objects and their authentication would be handled outside the SQL environment. The co-existence of ROLE and USER objects results in complexity. With the addition of ROLE objects, there is no rationale, other than legacy support, for granting privileges to USER objects directly. It is better to create roles and grant privileges to them, then grant the roles to USER objects.

The Standard effectively defines a special ROLE, named PUBLIC. All authorization have the PUBLIC role, which cannot be removed from them. Therefore any access right assigned to the PUBLIC role applies to all authorizations in the database. For many simple databases, it is adequate to create a single, non-admin user, then assign access rights to the pre-existing PUBLIC role. Access to INFORMATION_SCHEMA views is granted to PUBLIC, therefore these views are accessible to all. However, the contents of each view depends on the ROLE or USER (AUTHORIZATION) that is in force while accessing the view.

Each schema has a single AUTHORIZATION. This is commonly known as the *owner* of the schema. All the objects in the schema inherit the schema owner. The schema owner can add objects to the schema, drop them or alter them.

By default, the objects in a schema can only be accessed by the schema owner. The schema owner can grant access rights on the objects to other users or roles.

authorization identifier

authorization identifier

```
<authorization identifier> ::= <role name> | <user name>
```

Authorization identifiers share the same name-space within the database. The same name cannot be used for a USER and a ROLE.

Built-In Roles and Users

There are some pre-defined roles in each database; some defined by the SQL Standard, some by HyperSQL. These roles can be assigned to users (directly or via other, user-defined roles). In addition, there is the default initial user, SA, created with each new database.

PUBLIC

the PUBLIC role

The role that is assigned to all authorizations (roles and users) in the database. This role has access rights to all objects in the INFORMATION_SCHEMA. Any roles or rights granted to this role, are in effect granted to all users of the database.

_SYSTEM

the _SYSTEM role

This role is the authorization for the pre-defined (system) objects in the database, including the INFORMATION_SCHEMA. This role cannot be assigned to any authorization.

DBA

the DBA role (HyperSQL-specific)

This is a special role in HyperSQL. A user that has this role can perform all possible administrative tasks on the database. The DBA role can also act as a proxy for all the roles and users in the database. This means it can do everything the authorization for a schema can do, including dropping the schema or its objects, or granting rights on the schema objects to a grantee.

CREATE_SCHEMA

the CREATE_SCHEMA role (HyperSQL-specific)

An authorization that has this role, can create schemas. The DBA authorization has this role and can grant it to other authorizations.

CHANGE_AUTHORIZATION

the CHANGE_AUTHORIZATION role (HyperSQL-specific)

A user that has this role, can change the authorization for the current session to another user. The DBA authorization has this role and can grant it to other authorizations.

SA

the SA user (HyperSQL-specific)

This user is automatically created with a new database and has the DBA role. Initially, the password for this user is an empty string. After connecting to the new database as this user, it is possible to change the password, create other users and created new schema objects. The SA user can be dropped by another user that has the DBA role.

Access Rights

By default, the objects in a schema can only be accessed by the schema owner. But the schema owner can grant privileges (access rights) on the objects to other users or roles.

Things can get far more complex, because the grant of privileges can be made WITH GRANT OPTION. In this case, the role or user that has been granted the privilege can grant the privilege to other roles and users.

Privileges can also be revoked from users or roles.

The statements for granting and revoking privileges normally specify which privileges are granted or revoked. However, there is a shortcut, ALL PRIVILEGES, which means all the privileges that the `<grantor>` has on the schema object. The `<grantor>` is normally the CURRENT_USER of the session that issues the statement.

The user or role that is granted privileges is referred to as `<grantee>` for the granted privileges.

Table

For tables, including views, privileges can be granted with different degrees of granularity. It is possible to grant a privilege on all columns of a table, or on specific columns of the table.

The DELETE privilege applies to the table, rather than its columns. It applies to all DELETE statements.

The SELECT, INSERT and UPDATE privileges may apply to all columns or to individual columns. These privileges determine whether the `<grantee>` can execute SQL data statements on the table.

The SELECT privilege designates the columns that can be referenced in SELECT statements, as well as the columns that are read in a DELETE or UPDATE statement, including the search condition.

The INSERT privilege designates the columns into which explicit values can be inserted. To be able to insert a row into the table, the user must therefore have the INSERT privilege on the table, or at least all the columns that do not have a default value.

The UPDATE privilege simply designates the table or the specific columns that can be updated.

The REFERENCES privilege allows the `<grantee>` to define a FOREIGN KEY constraint on a different table, which references the table or the specific columns designated for the REFERENCES privilege.

The TRIGGER privilege allows adding a trigger to the table.

Sequence, Type, Domain, Character Set, Collation, Transliteration,

For these objects, only USAGE can be granted. The USAGE privilege is needed when object is referenced directly in an SQL statement.

Routine

For routines, including procedures or functions, only EXECUTE privilege can be granted. This privilege is needed when the routine is used directly in an SQL statement.

Other Objects

Other objects such as constraints and assertions are not used directly and there is no grantable privilege that refers to them.

Statements for Authorization and Access Control

The statements listed below allow creation and destruction of USER and ROLE objects. The GRANT and REVOKE statements allow roles to be assigned to other roles or to users. The same statements are also used in a different form to assign privileges on schema objects to users and roles.

CREATE USER

user definition (HyperSQL)

```
<user definition> ::= CREATE USER <user name> PASSWORD <password> [ ADMIN ]
```

Define a new user and its password. `<user name>` is an SQL identifier. If it is double-quoted it is case-sensitive, otherwise it is turned to uppercase. `<password>` is a string enclosed with single quote characters and is case-sensitive. If `ADMIN` is specified, the DBA role is granted to the new user. Only a user with the DBA role can execute this statement.

DROP USER

drop user statement (HyperSQL)

```
<drop user statement> ::= DROP USER <user name>
```

Drop (destroy) an existing user. If the specified user is the authorization for a schema, the schema is destroyed. Only a user with the DBA role can execute this statement.

ALTER USER ... SET PASSWORD

set the password for a user (HyperSQL)

```
<alter user set password statement> ::= ALTER USER <user name> SET PASSWORD
<password>
```

Change the password of an existing user. `` is an SQL identifier. If it is double-quoted it is case-sensitive, otherwise it is turned to uppercase. `<password>` is a string enclosed with single quote characters and is case-sensitive. Only a user with the DBA role can execute this command.

ALTER USER ... SET INITIAL SCHEMA

set the initial schema for a user (HyperSQL)

```
<alter user set initial schema statement> ::= ALTER USER <user name> SET INITIAL
SCHEMA <schema name> | DEFAULT
```

Change the initial schema for a user. The initial schema is the schema used by default for SQL statements issued during a session. If `DEFAULT` is used, the default initial schema for all users is used as the initial schema for the user. The SET SCHEMA command allows the user to change the schema for the duration of the session. Only a user with the DBA role can execute this statement.

SET PASSWORD

set password statement (HyperSQL)

```
<set password statement> ::= SET PASSWORD <password>
```

Set the password for the current user. `<password>` is a string enclosed with single quote characters and is case-sensitive.

SET INITIAL SCHEMA

set the initial schema for the current user (HyperSQL)

```
<set initial schema statement> ::= SET INITIAL SCHEMA <schema name> | DEFAULT
```

Change the initial schema for the current user. The initial schema is the schema used by default for SQL statements issued during a session. If `DEFAULT` is used, the default initial schema for all users is used as the initial schema for the current user. The separate SET SCHEMA command allows the user to change the schema for the duration of the session. See also the Sessions and Transactions chapter.

SET DATABASE DEFAULT INITIAL SCHEMA

set the default initial schema for all users (HyperSQL)

```
<set database default initial schema statement> ::= SET DATABASE DEFAULT INITIAL
SCHEMA <schema name>
```

Sets the initial schema for new users. This schema can later be changed with the `<set initial schema statement>` command.

CREATE ROLE

role definition

```
<role definition> ::= CREATE ROLE <role name> [ WITH ADMIN <grantor> ]
```

Defines a new role. Initially the role has no rights, except those of the PUBLIC role. Only a user with the DBA role can execute this command.

DROP ROLE

drop role statement

```
<drop role statement> ::= DROP ROLE <role name>
```

Drop (destroy) a role. If the specified role is the authorization for a schema, the schema is destroyed. Only a user with the DBA role can execute this statement.

GRANTED BY

grantor determination

```
GRANTED BY <grantor>
```

```
<grantor> ::= CURRENT_USER | CURRENT_ROLE
```

The authorization that is granting or revoking a role or privileges. The optional `GRANTED BY <grantor>` clause can be used in various statements that perform GRANT or REVOKE actions. If the clause is not used, the authorization is CURRENT_USER. Otherwise, it is the specified authorization.

GRANT

grant privilege statement

```
<grant privilege statement> ::= GRANT <privileges> TO <grantee> [ { <comma>
<grantee> }... ] [ WITH GRANT OPTION ] [ GRANTED BY <grantor> ]
```

Assign privileges on schema objects to roles or users. Each `<grantee>` is a role or a user. If `[ WITH GRANT OPTION ]` is specified, then the `<grantee>` can assign the privileges to other `<grantee>` objects.

```
<privileges> ::= <object privileges> ON <object name>
```

```
<object name> ::= [ TABLE ] <table name> | DOMAIN <domain name> | COLLATION
<collation name> | CHARACTER SET <character set name> | TRANSLATION <transliter-
ation name> | TYPE <user-defined type name> | SEQUENCE <sequence generator name>
| <specific routine designator> | ROUTINE <routine name> | FUNCTION <function
name> | PROCEDURE <procedure name>
```

```
<object privileges> ::= ALL PRIVILEGES | <action> [ { <comma> <action> }... ]
```

```
<action> ::= SELECT | SELECT <left paren> <privilege column list> <right paren>
| DELETE | INSERT [ <left paren> <privilege column list> <right paren> ] | UPDATE
[ <left paren> <privilege column list> <right paren> ] | REFERENCES [ <left
paren> <privilege column list> <right paren> ] | USAGE | TRIGGER | EXECUTE
```

```
<privilege column list> ::= <column name list>
```

```
<grantee> ::= PUBLIC | <authorization identifier>
```

The `<object privileges>` that can be used depend on the type of the `<object name>`. These are discussed in the previous section. For a table, if `<privilege column list>` is not specified, then the privilege is granted on the table, which includes all of its columns and any column that may be added to it in the future. For routines, the name of the routine can be specified in two ways, either as the generic name as the specific name. HyperSQL allows referencing all overloaded versions of a routine at the same time, using its name. This differs from the SQL Standard which requires the use of `<specific routine designator>` to grant privileges separately on each different signature of the routine.

Each `<grantee>` is the name of a role or a user. Examples of GRANT statement are given below:

```
GRANT ALL ON SEQUENCE aSequence TO roleOrUser
```

```
GRANT SELELCT ON aTable TO roleOrUser
GRANT SELECT, UPDATE ON aTABLE TO roleOrUser1, roleOrUser2
GRANT SELECT(columnA, columnB), UPDATE(columnA, columnB) ON TABLE aTable TO roleOrUser
GRANT EXECUTE ON SPECIFIC ROUTINE aroutine_1234 TO rolOrUser
```

As mentioned in the general discussion, it is better to define a role for the collection of all the privileges required by an application. This role is then granted to any user. If further changes are made to the privileges of this role, they are automatically reflected in all the users that have the role.

GRANT

grant role statement

```
<grant role statement> ::= GRANT <role name> [ { <comma> <role name> }... ]
TO <grantee> [ { <comma> <grantee> }... ] [ WITH ADMIN OPTION ] [ GRANTED BY
<grantor> ]
```

Assign roles to roles or users. One or more roles can be assigned to one or more `<grantee>` objects. A `<grantee>` is a user or a role. If the `[ WITH ADMIN OPTION ]` is specified, then each `<grantee>` can grant the newly assigned roles to other grantees. An example of user and role creation with grants is given below:

```
CREATE USER appuser
CREATE ROLE approle
GRANT approle TO appuser
GRANT SELECT, UPDATE ON TABLE atable TO approle
GRANT USAGE ON SEQUENCE asequence to approle
GRANT EXECUTE ON ROUTINE aroutine TO approle
```

REVOKE privilege

revoke statement

```
<revoke privilege statement> ::= REVOKE [ GRANT OPTION FOR ] <privileges> FROM
<grantee> [ { <comma> <grantee> }... ] [ GRANTED BY <grantor> ] RESTRICT | CASCADE
```

Revoke privileges from a user or role.

REVOKE role

revoke role statement

```
<revoke role statement> ::= REVOKE [ ADMIN OPTION FOR ] <role revoked> [ { <comma>
<role revoked> }... ] FROM <grantee> [ { <comma> <grantee> }... ] [ GRANTED
BY <grantor> ] RESTRICT | CASCADE
```

```
<role revoked> ::= <role name>
```

Revoke a role from users or roles.

Chapter 7. Data Access and Change

Fred Toussi, The HSQL Development Group

$Revision: 3601 $

Copyright 2010 Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2010-05-31 20:17:47 -0400 (Mon, 31 May 2010) $

Overview

HyperSQL data access and data change statements are fully compatible with the latest SQL:2008 Standard. There are a few extensions and some relaxation of rules, but these do not affect statements that are written to the Standard syntax. There is full support for classic SQL, as specified by SQL-92, and many enhancements added in later versions of the standard.

Cursors And Result Sets

An SQL statement can executed in two ways. One way is to use the `java.sql.Statement` interface. The Statement object can be reused to execute completely different SQL statements. Alternatively a `PreparedStatment` can be used to execute an SQL statement repeatedly, and the statements can be parameterized. Using either form, if the SQL statement is a query expression, a `ResultSet` is returned.

In SQL, when a query expression (SELECT or similar SQL statement) is executed, an ephemeral table is created. When this table is returned to the application program, it is returned as a result set, which is accessed row-by-row by a cursor. A JDBC `ResultSet` represents an SQL result set and its cursor.

The minimal definition of a cursor is a list of rows with a position that can move forward. Some cursors also allow the position to move backwards or jump to any position in the list.

An SQL cursor has several attributes. These attributes depend on the query expression. Some of these attributes can be overridden by specifying qualifiers in the SQL statement or by specifying values for the parameters of the JDBC `Statement` or `PreparedStatement`.

Columns and Rows

The columns of the rows of the result set are determined by the query expression. The number of columns and the type and name characteristics of each column are known when the query expression is compiled and before its execution. This metadata information remains constant regardless of changes to the contents of the tables used in the query expression. The metadata for the JDBC `ResultSet` is in the form of a `ResultSetMetaData` object. Various methods of the `ResultSetMetaData` interface return different properties of each column of the `ResultSet`.

A result set may contain 0 or more rows. The rows are determined by the execution of the query expression.

The `setMaxRows(int)` method of JDBC `Statement` allows limiting the number of rows returned by the statement. This limit is conceptually applied after the result has been built, and the excess rows are discarded.

Navigation

A cursor is either scrollable or not. Scrollable cursors allow accessing rows by absolute or relative positioning. No-scroll cursors only allow moving to the next row. The cursor can be optionally declared with the SQL qualifiers

SCROLL, or NO SCROLL. The JDBC statement parameter can be specified as: TYPE_FORWARD_ONLY and TYPE_SCROLL_INSENSITIVE. The JDBC type TYPE_SCROLL_SENSITIVE is not supported by HSQLDB.

The default is NO SCROLL or TYPE_FORWARD_ONLY.

When a JDBC `ResultSet` is opened, it is positioned before the first row. Using the `next()` method the position is moved to the first row. While the `ResultSet` is positioned on a row, various getter methods can be used to access the columns of the row.

Updatability

The result returned by some query expressions is updatable. HSQLDB supports core SQL updatability features, plus some enhancements from the SQL optional features.

A query expression is updatable if it is a SELECT from a single underlying base table (or updatable view) either directly or indirectly. A SELECT statement featuring DISTINCT or GROUP BY or FETCH, LIMIT, OFFSET is not updatable. In an updatable query expression, one or more columns are updatable. An updatable column is a column that can be traced directly to the underlying table. Therefore, columns that contain expressions are not updatable. Examples of updatable query expressions are given below. The view V is updatable when its query expression is updatable. The SELECT statement from this view is also updatable:

```
SELECT A, B FROM T WHERE C > 5
SELECT A, B FROM (SELECT * FROM T WHERE C > 10) AS TT WHERE TT.B <10
CREATE VIEW V(X,Y) AS SELECT A, B FROM T WHERE C > 0 AND B < 10
SELECT X FROM V WHERE Y = 5
```

If a cursor is declared with the SQL qualifier, `FOR UPDATE OF <column name list>`, then only the stated columns in the result set become updatable. If any of the stated columns is not actually updatable, then the cursor declaration will not succeed.

If the SQL qualifier, FOR UPDATE is used, then all the updatable columns of the result set become updatable.

If a cursor is declared with FOR READ ONLY, then it is not updatable.

In HSQLDB, if FOR READ ONLY or FOR UPDATE is not used then all the updatable columns of the result set become updatable. This relaxes the SQL standard rule that in this case limits updatability to only simply updatable SELECT statements (where all columns are updatable).

In JDBC, CONCUR_READ_ONLY or CONCUR_UPDATABLE can be specified for the `Statement` parameter. CONCUR_UPDATABLE is required if the returning ResultSet is to be updatable. If CONCUR_READ_ONLY, which is the default, is used, then even an updatable ResultSet becomes read-only.

When a `ResultSet` is updatable, various setter methods can be used to modify the column values. The names of the setter methods begin with "update". After all the updates on a row are done, the `updateRow()` method must be called to finalise the row update.

An updatable `ResultSet` may or may not be insertable-into. In an insertable `ResultSet`, all columns of the result are updatable and any column of the base table that is not in the result must be a generated column or have a default value.

In the `ResultSet` object, a special pseudo-row, called the insert row, is used to populate values for insertion into the `ResultSet` (and consequently, into the base table). The setter methods must be used on all the columns, followed by a call to `insertRow()`.

Individual rows from all updatable result sets can be deleted one at a time. The `deleteRow()` is called when the `ResultSet` is positioned on a row.

While using an updatable ResultSet to modify data, it is recommended not to change the same data using another ResultSet and not to execute SQL data change statements that modify the same data.

Sensitivity

The sensitivity of the cursor relates to visibility of changes made to the data by the same transaction but without using the given cursor. While the result set is open, the same transaction may use statements such as INSERT or UPDATE, and change the data of the tables from which the result set data is derived. A cursor is SENSITIVE if it reflects those changes. It is INSENSITIVE if it ignores such changes. It is ASENSITIVE if behaviour is implementation dependent.

The SQL default is ASENSITIVE, i.e., implantation dependent.

In HSQLDB all cursors are INSENSITIVE. They do not reflect changes to the data made by other statements.

Holdability

A cursor is holdable if the result set is not automatically closed when the current transaction is committed. Holdability can be specified in the cursor declaration using the SQL qualifiers WITH HOLD or WITHOUT HOLD.

In JDBC, hodability is specified using either of the following values for the Statement parameter: HOLD_CURSORS_OVER_COMMIT, or CLOSE_CURSORS_AT_COMMIT.

The SQL default is WITHOUT HOLD.

The JDBC default for HSQLDB result sets is WITH HOLD for read-only result sets and WITHOUT HOLD for updatable result sets.

If the holdability of a `ResultSet` is specified in a conflicting manner in the SQL statement and the JDBC `Statement` object, the JDBC setting takes precedence.

Autocommit

The autocommit property of a connection is a feature of JDBC and ODBC and is not part of the SQL Standard. In autocommit mode, all transactional statements are followed by an implicit commit. In autocommit mode, all `ResultSet` objects are read-only and holdable.

JDBC Overview

The JDBC settings, ResultSet.CONCUR_READONLY and ResultSet.CONCUR_UPDATABLE are the alternatives for read-only or updatability. The default is ResultSet.CONCUR_READONLY.

The JDBC settings, ResultSet.TYPE_FORWARD_ONLY, ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.TYPE_SCROLL_SENSITIVE are the alternatives for both scrollability (navigation) and sensitivity. HyperSQL does not support ResultSet.TYPE_SCROLL_SENSITIVE. The two other alternatives can be used for both updatable and read-only result sets.

The JDBC settings ResultSet.CLOSE_CURSORS_AT_COMMIT and ResultSet.HOLD_CURSORS_OVER_COMMIT are the alternatives for the lifetime of the result set. The default is ResultSet.CLOSE_CURSORS_AT_COMMIT. The other setting can only be used for read-only result sets.

Examples of creating statements for updatable result sets are given below:

```
Connection c = newConnection();
Statement st;
```

```
c.setAutoCommit(false);
st = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
st = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

JDBC Parameters

When a JDBC PreparedStatement or CallableStatement is used with an SQL statement that contains dynamic parameters, the data types of the parameters are resolved and determined by the engine when the statement is prepared. The SQL Standard has detailed rules to determine the data types and imposes limits on the maximum length or precision of the parameter. HyperSQL applies the standard rules with two exceptions for parameters with String and BigDecimal Java types. HyperSQL ignores the limits when the parameter value is set, and only enforces the necessary limits when the PreparedStatement is executed. In all other cases, parameter type limits are checked and enforce when the parameter is set.

In the example below the setString() calls do not raise an exception, but one of the execute() statements does.

```
// table definition: CREATE TABLE T (NAME VARCHAR(12), ...)
Connection c = newConnection();
PreparedStatement st = c.prepareStatement("SELECT * FROM T WHERE NAME = ?");
// type of the parameter is VARCHAR(12), which limits length to 12 characters
st.setString(1, "Eyjafjallajokull"); // string is longer than type, but no exception is raised
 here
set.execute(); // executes with no exception and does not find any rows
// but if an UPDATE is attempted, an exception is raised
st = c.prepareStatement("UPDATE T SET NAME = ? WHERE ID = 10");
st.setString(1, "Eyjafjallajokull"); // string is longer than type, but no exception is raised
 here
st.execute(); // exception is thrown when HyperSQL checks the value for update
```

All of the above also applies to setting the values in new and updated rows in updatable ResultSet objects.

JDBC parameters can be set with any compatible type, as supported by the JDBC specification. For CLOB and BLOB types, you can use streams, or create instances of BLOB or CLOB before assigning them to the parameters. You can even use CLOB or BLOB objects returned from connections to other RDBMS servers. The Connection.createBlob() and createClob() methods can be used to create the new LOBs. For very large LOB's the stream methods are preferable as they use less memory.

For array parameters, you must use a java.sql.Array object that contains the array elements before assigning to JDBC parameters. The Connection.createArrayOf(...) method can be used to create a new object, or you can use an Array returned from connections to other RDBMS servers.

JDBC Returned Values

The methods of the JDBC ResultSet interface can be used to return values and to convert value to different types as supported by the JDBC specification.

When a CLOB and BLOB object is returned from a ResultSet, no data is transferred until the data is read by various methods of java.sql.CLOB and java.sql.BLOB. Data is streamed in large blocks to avoid excessive memory use.

Array objects are returned as instances of java.sql.Array.

Syntax Elements

The syntax elements that can be used in data access and data change statements are described in this section. The SQL Standard has a very extensive set of definitions for these elements. The BNF definitions given here are sometimes simplified.

Literals

Literals are used to express constant values. The general type of a literal is known by its format. The specific type is based on conventions.

unicode escape elements

unicode escape elements

```
<Unicode  escape  specifier>  ::=  [  UESCAPE  <quote><Unicode  escape
character><quote> ]
```

```
<Unicode escape value> ::= <Unicode 4 digit escape value> | <Unicode 6 digit
escape value> | <Unicode character escape value>
```

```
<Unicode   4   digit   escape   value>   ::=   <Unicode   escape
character><hexit><hexit><hexit><hexit>
```

```
<Unicode 6 digit escape value> ::= <Unicode escape character><plus sign>
<hexit><hexit><hexit><hexit><hexit><hexit>
```

```
<Unicode character escape value> ::= <Unicode escape character><Unicode escape
character>
```

```
<Unicode escape character> ::= a single character than a <hexit> (a-f, A-F,
0-9), <plus sign>, <quote>, <double quote>, or <white space>
```

character literal

character literal

```
<character string literal> ::= [ <introducer><character set specification> ]
<quote> [ <character representation>... ] <quote> [ { <separator> <quote>
[ <character representation>... ] <quote> }... ]
```

```
<introducer> ::= <underscore>
```

```
<character representation> ::= <nonquote character> | <quote symbol>
```

```
<nonquote character> ::= any character apart from the quote symbol.
```

```
<quote symbol> ::= <quote><quote>
```

```
<national character string literal> ::= N <quote> [ <character representa-
tion>... ] <quote> [ { <separator> <quote> [ <character representation>... ]
<quote> }... ]
```

```
<Unicode character string literal> ::= [ <introducer><character set specifica-
tion> ] U<ampersand><quote> [ <Unicode representation>... ] <quote> [ { <sepa-
rator> <quote> [ <Unicode representation>... ] <quote> }... ] <Unicode escape
specifier>
```

```
<Unicode representation> ::= <character representation> | <Unicode escape value>
```

The type of a character literal is CHARACTER. The length of the string literal is the character length of the type. If the quote character is used in a string, it is represented with two quote characters. Long literals can be divided into multiple quoted strings, separated with a space or end-of-line character.

Unicode literals start with U& and can contain ordinary characters and unicode escapes. A unicode escape begins with the backslash (\ ) character and is followed by four hexadecimal characters which specify the character code.

Example of character literals are given below:

```
'a literal'  ' string seperated'  ' into parts'
'a string''s literal form with quote character'
U&'Unicode string with Greek delta \0394 and phi \03a6 letters'
```

binary literal

binary literal

```
<binary string literal> ::= X <quote> [ <space>... ] [ { <hexit> [ <space>... ]
<hexit> [ <space>... ] }... ] <quote> [ { <separator> <quote> [ <space>... ]
[ { <hexit> [ <space>... ] <hexit> [ <space>... ] }... ] <quote> }... ]
```

```
<hexit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f
```

The type of a binary literal is BINARY. The octect length of the binary literal is the length of the type. Case-insensitive hexadecimal characters are used in the binary string. Each pair of characters in the literal represents a byte in the binary string. Long literals can be divided into multiple quoted strings, separated with a space or end-of-line character.

```
X'1abACD34' 'Af'
```

bit literal

bit literal

```
<bit string literal> ::= B <quote> [ <bit> ... ] <quote> [ { <separator> <quote>
[ <bit>... ] <quote> }... ]
```

```
<bit> ::= 0 | 1
```

The type of a binary literal is BIT. The bit length of the bit literal is the length of the type. Digits 0 and 1 are used to represent the bits. Long literals can be divided into multiple quoted strings, separated with a space or end-of-line character.

```
B'10001001' '00010'
```

numeric literal

numeric literal

```
<signed numeric literal> ::= [ <sign> ] <unsigned numeric literal>
```

```
<unsigned numeric literal> ::= <exact numeric literal> | <approximate numeric
literal>
```

```
<exact numeric literal> ::= <unsigned integer> [ <period> [ <unsigned integer> ] ]
| <period> <unsigned integer>
```

```
<sign> ::= <plus sign> | <minus sign>
```

```
<approximate numeric literal> ::= <mantissa> E <exponent>
```

```
<mantissa> ::= <exact numeric literal>
```

```
<exponent> ::= <signed integer>

<signed integer> ::= [ <sign> ] <unsigned integer>

<unsigned integer> ::= <digit>...
```

The type of an exact numeric literal without a decimal point is INTEGER, BIGINT, or DECIMAL, depending on the value of the literal (the smallest type that can represent the value is the type).

The type of an exact numeric literal with a decimal point is DECIMAL. The precision of a decimal literal is the total number of digits of the literal. The scale of the literal is the total number of digits to the right of the decimal point.

The type of an approximate numeric literal is DOUBLE. An approximate numeric literal always includes the mantissa and exponent, separated by E.

```
12
34.35
+12E-2
```

boolean literal

boolean literal

```
<boolean literal> ::= TRUE | FALSE | UNKNOWN
```

The boolean literal is one of the specified keywords.

datetime and interval literal

datetime and interval literal

```
<datetime literal> ::= <date literal> | <time literal> | <timestamp literal>

<date literal> ::= DATE <date string>

<time literal> ::= TIME <time string>

<timestamp literal> ::= TIMESTAMP <timestamp string>

<date string> ::= <quote> <unquoted date string> <quote>

<time string> ::= <quote> <unquoted time string> <quote>

<timestamp string> ::= <quote> <unquoted timestamp string> <quote>

<time zone interval> ::= <sign> <hours value> <colon> <minutes value>

<date value> ::= <years value> <minus sign> <months value> <minus sign> <days value>

<time value> ::= <hours value> <colon> <minutes value> <colon> <seconds value>

<interval literal> ::= INTERVAL [ <sign> ] <interval string> <interval qualifier>

<interval string> ::= <quote> <unquoted interval string> <quote>

<unquoted date string> ::= <date value>
```

```
<unquoted time string> ::= <time value> [ <time zone interval> ]

<unquoted timestamp string> ::= <unquoted date string> <space> <unquoted time
string>

<unquoted interval string> ::= [ <sign> ] { <year-month literal> | <day-time
literal> }

<year-month literal> ::= <years value> [ <minus sign> <months value> ] | <months
value>

<day-time literal> ::= <day-time interval> | <time interval>

<day-time interval> ::= <days value> [ <space> <hours value> [ <colon> <minutes
value> [ <colon> <seconds value> ] ] ]

<time interval> ::= <hours value> [ <colon> <minutes value> [ <colon> <seconds
value> ] ] | <minutes value> [ <colon> <seconds value> ] | <seconds value>

<years value> ::= <datetime value>

<months value> ::= <datetime value>

<days value> ::= <datetime value>

<hours value> ::= <datetime value>

<minutes value> ::= <datetime value>

<seconds value> ::= <seconds integer value> [ <period> [ <seconds fraction> ] ]

<seconds integer value> ::= <unsigned integer>

<seconds fraction> ::= <unsigned integer>

<datetime value> ::= <unsigned integer>
```

The type of a datetime or interval type is specified in the literal. The fractional second precision is the number of digits in the fractional part of the literal. Details are described in the SQL Language chapter

```
DATE '2008-08-08'
TIME '20:08:08'
TIMESTAMP '2008-08-08 20:08:08.235'

INTERVAL '10' DAY
INTERVAL -'08:08' MINUTE TO SECOND
```

References, etc.

References are identifier chains, which can be a single identifiers or identifiers chains composed of single identifiers chained together with the period symbol.

identifier chain

identifier chain

```
<identifier chain> ::= <identifier> [ { <period> <identifier> }... ]

<basic identifier chain> ::= <identifier chain>
```

A period-separated chain of identifiers. The identifiers in an identifier chain can refer to database objects in a hierarchy. The possible hierarchies are as follows. In each hierarchy, elements from the start or the end can be missing, but the order of elements cannot be changed.

catalog, schema, database object

catalog, schema, table, column

correlation name, column

Examples of identifier chain are given below:

```
SELECT MYCAT.MYSCHEMA.MYTABLE.MYCOL FROM MYCAT.MYSCHEMA.MYTABLE
DROP TABLE MYCAT.MYSCHEMA.MYTABLE CASCADE
ALTER SEQUENCE MYCAT.MYSCHEMA.MYSEQUENCE RESTART WITH 100
```

column reference

column reference

```
<column reference> ::= <basic identifier chain> | MODULE <period> <qualified
identifier> <period> <column name>
```

Reference a column or a routine variable.

SQL parameter reference

SQL parameter reference

```
<SQL parameter reference> ::= <basic identifier chain>
```

Reference an SQL routine parameter.

contextually typed value specification

contextually typed value specification

```
<contextually typed value specification> ::= <null specification> | <default
specification>
```

```
<null specification> ::= NULL
```

```
<default specification> ::= DEFAULT
```

Specify a value whose data type or value is inferred from its context. DEFAULT is used for assignments to table columns that have a default value, or to table columns that are generated either as an IDENTITY value or as an expression. NULL can be used only in a context where the type of the value is known. For example, a NULL can be assigned to a column of the table in an INSERT or UPDATE statement, because the type of the column is known. But if NULL is used in a SELECT list, it must be used in a CAST statement.

Value Expression

Value expression is a general name for all expressions that return a value. Different types of expressions are allowed in different contexts.

value expression primary

value expression primary

```
<value expression primary> ::= <parenthesized value expression> | <nonparenthe-
sized value expression primary>
```

```
<parenthesized value expression> ::= <left paren> <value expression> <right
paren>
```

```
<nonparenthesized value expression primary> ::= <unsigned value specification>
| <column reference> | <set function specification> | <scalar subquery> |
<case expression> | <cast specification> | <next value expression> | <routine
invocation>
```

Specify a value that is syntactically self-delimited.

value specification

value specification

```
<value specification> ::= <literal> | <general value specification>
```

```
<unsigned value specification> ::= <unsigned literal> | <general value speci-
fication>
```

```
<target specification> ::= <host parameter specification> | <SQL parameter
reference> | <column reference> | <dynamic parameter specification>
```

```
<simple target specification> ::= <host parameter specification> | <SQL parameter
reference> | <column reference> | <embedded variable name>
```

```
<host parameter specification> ::= <host parameter name> [ <indicator parame-
ter> ]
```

```
<dynamic parameter specification> ::= <question mark>
```

Specify one or more values, host parameters, SQL parameters, dynamic parameters, or host variables.

row value expression

row value expression

```
<row value expression> ::= <row value special case> | <explicit row value
constructor>
```

```
<row value predicand> ::= <row value special case> | <row value constructor
predicand>
```

```
<row value special case> ::= <nonparenthesized value expression primary>
```

```
<explicit row value constructor> ::= <left paren> <row value constructor element>
<comma> <row value constructor element list> <right paren> |
```

```
ROW <left paren> <row value constructor element list> <right paren> | <row
subquery>
```

Specify a row consisting of one or more elements. A comma separated list of expressions, enclosed in brackets, with the optional keyword ROW. In SQL, a row containing a single element can often be used where a single value is expected.

set function specification

set function specification

```
<set function specification> ::= <aggregate function> | <grouping operation>

<grouping operation> ::= GROUPING <left paren> <column reference> [ { <comma>
<column reference> }... ] <right paren>
```

Specify a value derived by the application of a function to an argument. Early releases of HyperSQL 2.0 do not support `<grouping operation>`.

COALESCE

coalesce expression

```
<coalesce expression> := COALESCE <left paren> <value expression> { <comma>
<value expression> }... <right paren>
```

Replace null values with another value. The coalesce expression has two or more instances of <value expression>. If the first <value expression> evaluates to a non-null value, it is returned as the result of the coalesce expression. If it is null, the next `<value expression>` is evaluated and if it evaluates to a non-non value, it is returned, and so on.

The type of the return value of a COALESCE expression is the aggregate type of the types of all the `<value expression>` instances. Therefore, any value returned is implicitly cast to this type. HyperSQL also features built-in functions with similar functionality.

NULLIF

nullif expression

```
<nullif expression> := NULLIF <left paren> <value expression> <comma> <value
expression> <right paren>
```

Return NULL if two values are equal. If the result of the first `<value expression>` is not equal to the result of the second, then it is returned, otherwise NULL is returned. The type of the return value is the type of the first `<value expression>`.

```
SELECT i, NULLIF(n, 'not defined') FROM t
```

CASE

case specification

```
<case specification> ::= <simple case> | <searched case>

<simple case> ::= CASE <case operand> <simple when clause>... [ <else clause> ]
END

<searched case> ::= CASE <searched when clause>... [ <else clause> ] END

<simple when clause> ::= WHEN <when operand list> THEN <result>

<searched when clause> ::= WHEN <search condition> THEN <result>

<else clause> ::= ELSE <result>

<case operand> ::= <row value predicand> | <overlaps predicate part 1>

<when operand list> ::= <when operand> [ { <comma> <when operand> }... ]
```

```
<when operand> ::= <row value predicand> | <comparison predicate part 2> |
<between predicate part 2> | <in predicate part 2> | <character like predicate
part 2> | <octet like predicate part 2> | <similar predicate part 2> | <regex like
predicate part 2> | <null predicate part 2> | <quantified comparison predicate
part 2> | <match predicate part 2> | <overlaps predicate part 2> | <distinct
predicate part 2>

<result> ::= <result expression> | NULL

<result expression> ::= <value expression>
```

Specify a conditional value. The result of a case expression is always a value. All the values introduced with THEN must be of the same type.

An (simple) example of the CASE statement is given below. It returns 'Britain', 'Germany', or 'Other country' depending on the value of dialcode'

```
CASE dialcode WHEN 44 THEN 'Britain' WHEN 49 THEN 'Germany' ELSE 'Other country' END
```

The case statement can be far more complex and involve several conditions.

CAST

cast specification

```
<cast specification> ::= CAST <left paren> <cast operand> AS <cast target>
<right paren>

<cast operand> ::= <value expression> | <implicitly typed value specification>

<cast target> ::= <domain name> | <data type>
```

Specify a data conversion. Data conversion takes place automatically among variants of a general type. For example numeric values are freely converted from one type to another in expressions.

Explicit type conversion is necessary in two cases. One case is to determine the type of a NULL value. The other case is to force conversion for special purposes. Values of data types can be cast to a character type. The exception is BINARY and OTHER types. The result of the cast is the literal expression of the value. Conversely, a value of a character type can be converted to another type if the character value is a literal representation of the value in the target type. Special conversions are possible between numeric and interval types, which are described in the section covering interval types.

The examples below show examples of cast with their result:

```
CAST (NULL AS TIMESTAMP)
CAST ('   199  ' AS INTEGER) = 199
CAST ('tRue ' AS BOOLEAN) = TRUE
CAST (INTERVAL '2' DAY AS INTEGER) = 2
CAST ('1992-04-21' AS DATE) = DATE '1992-04-21'
```

NEXT VALUE FOR

next value expression

```
<next value expression> ::= NEXT VALUE FOR <sequence generator name>
```

Return the next value of a sequence generator. This expression can be used as a select list element in queries, or in assignments to table columns in data change statements. If the expression is used more than once in a single row that

is being evaluated, the same value is returned for each invocation. After evaluation of the particular row is complete, the sequence generator will return a different value from the old value. The new value is generated by the sequence generator by adding the increment to the last value it generated. In the example below:

```
INSERT INTO MYTABLE(COL1, COL2) VALUES 2, NEXT VALUE FOR MYSEQUENCE
```

value expression

value expression

```
<value expression> ::= <numeric value expression> | <string value expression>
| <datetime value expression> | <interval value expression> | <boolean value
expression> | <row value expression>
```

An expression that returns a value. The value can be a single value, or a row consisting more than one value.

numeric value expression

numeric value expression

```
<numeric value expression> ::= <term> | <numeric value expression> <plus sign>
<term> | <numeric value expression> <minus sign> <term>

<term> ::= <factor> | <term> <asterisk> <factor> | <term> <solidus> <factor>

<factor> ::= [ <sign> ] <numeric primary>

<numeric primary> ::= <value expression primary> | <numeric value function>
```

Specify a numeric value. The BNF indicates that `<asterisk>` and `<solidus>` (the operators for multiplication and division) have precedence over `<minus sign>` and `<plus sign>`.

numeric value function

numeric value function

```
<numeric value function> ::= <position expression> | <extract expression> |
<length expression> ...
```

Specify a function yielding a value of type numeric. The supported numeric value functions are listed and described in the Built In Functions chapter.

string value expression

string value expression

```
<string value expression> ::= <string concatenation> | <string factor>

<string factor> ::= <value expression primary> | <string value function>

<string concatenation> ::= <string value expression> <concatenation operator>
<string factor>

<concatenation operator> ::= ||
```

Specify a character string value, a binary string value, or a bit string value. The BNF indicates that a string value expression can be formed by concatenation of two or more `<value expression primary>`. The types of the

`<value expression primary>` elements must be compatible, that is, all must be string, or binary or bit string values.

character value function

string value function

```
<string value function> ::= ...
```

Specify a function that returns a character string or binary string. The supported character value functions are listed and described in the Built In Functions chapter.

datetime value expression

datetime value expression

```
<datetime value expression> ::= <datetime term> | <interval value expression>
<plus sign> <datetime term> | <datetime value expression> <plus sign> <interval
term> | <datetime value expression> <minus sign> <interval term>

<datetime term> ::= <datetime factor>

<datetime factor> ::= <datetime primary> [ <time zone> ]

<datetime primary> ::= <value expression primary> | <datetime value function>

<time zone> ::= AT <time zone specifier>

<time zone specifier> ::= LOCAL | TIME ZONE <interval primary>
```

Specify a datetime value. Details are described in the SQL Language chapter.

datetime value function

datetime value function

```
<datetime value function> ::= ...
```

Specify a function that returns a datetime value. The supported datetime value functions are listed and described in the Built In Functions chapter.

interval term

interval value expression

```
<interval value expression> ::= <interval term> | <interval value expression
1> <plus sign> <interval term 1> | <interval value expression 1> <minus sign>
<interval term 1> | <left paren> <datetime value expression> <minus sign>
<datetime term> <right paren> <interval qualifier>

<interval term> ::= <interval factor> | <interval term 2> <asterisk> <factor>
| <interval term 2> <solidus> <factor> | <term> <asterisk> <interval factor>

<interval factor> ::= [ <sign> ] <interval primary>

<interval primary> ::= <value expression primary> [ <interval qualifier> ] |
<interval value function>
```

```
<interval value expression 1> ::= <interval value expression>
```

```
<interval term 1> ::= <interval term>
```

```
<interval term 2> ::= <interval term>
```

Specify an interval value. Details are described in the SQL Language chapter.

interval absolute value function

interval value function

```
<interval value function> ::= <interval absolute value function>
```

```
<interval absolute value function> ::= ABS <left paren> <interval value expres-
sion> <right paren>
```

Specify a function that returns the absolute value of an interval. If the interval is negative, it is negated, otherwise
the original value is returned.

boolean value expression

boolean value expression

```
<boolean value expression> ::= <boolean term> | <boolean value expression> OR
<boolean term>
```

```
<boolean term> ::= <boolean factor> | <boolean term> AND <boolean factor>
```

```
<boolean factor> ::= [ NOT ] <boolean test>
```

```
<boolean test> ::= <boolean primary> [ IS [ NOT ] <truth value> ]
```

```
<truth value> ::= TRUE | FALSE | UNKNOWN
```

```
<boolean primary> ::= <predicate> | <boolean predicand>
```

```
<boolean predicand> ::= <parenthesized boolean value expression> | <nonparen-
thesized value expression primary>
```

```
<parenthesized boolean value expression> ::= <left paren> <boolean value ex-
pression> <right paren>
```

Specify a boolean value.

Predicates

Predicates are conditions with two sides and evaluate to a boolean value. The left side of the predicate, the `<row
value predicand>`, is the common element of all predicates. This element is a generalisation of both `<value
expression>`, which is a scalar, and of `<explicit row value constructor>`, which is a row. The two
sides of a predicate can be split in CASE statements where the `<row value predicand>` is part of multiple
predicates.

The number of fields in all `<row value predicand>` used in predicates must be the same and the types of the
fields in the same position must be compatible for comparison. If either of these conditions does not hold, an exception
is raised. The number of fields in a row is called the *degree*.

In many types of predicates (but not all of them), if the `<row value predicand>` evaluates to NULL, the result of the predicate is UNKNOWN. If the `<row value predicand>` has more than one element, and one or more of the fields evaluate to NULL, the result depends on the particular predicate.

comparison predicand

comparison predicate

```
<comparison predicate> ::= <row value predicand> <comp op> <row value predicand>

<comp op> ::= <equals operator> | <not equals operator> | <less than operator>
| <greater than operator> | <less than or equals operator> | <greater than or
equals operator>
```

Specify a comparison of two row values. If either `<row value predicand>` evaluates to NULL, the result of `<comparison predicate>` is UNKNOWN. Otherwise, the result is TRUE, FALSE or UNKNOWN.

If the *degree* of `<row value predicand>` is larger than one, comparison is performed between each field and the corresponding field in the other `<row value predicand>` from left to right, one by one.

When comparing two elements, if either field is NULL then the result is UNKNOWN.

For `<equals operator>`, if the result of comparison is TRUE for all field, the result of the predicate is TRUE. If the result of comparison is FALSE for one field, the result of predicate is FALSE. Otherwise the result is UNKNOWN.

The `<not equals operator>` is translated to `NOT (<row value predicand> = <row value predicand>)`.

The `<less than or equals operator>` is translated to `(<row value predicand> = <row value predicand>) OR (<row value predicand> < <row value predicand>)`. The `<greater than or equals operator>` is translated similarly.

For the `<less than operator>` and `<greater than operator>`, if two fields at a given position are equal, then comparison continues to the next field. Otherwise, the result of the last performed comparison is returned as the result of the predicate. This means that if the first field is NULL, the result is always UNKNOWN.

The logic that governs NULL values and UNKNOWN result is as follows: Suppose the NULL values were substituted by arbitrary real values. If substitution cannot change the result of the predicate, then the result is TRUE or FALSE, based on the existing non-NULL values, otherwise the result of the predicate is UNKNOWN.

The examples of comparison given below use literals, but the literals actually represent the result of evaluation of some expression.

```
((1, 2, 3, 4) = (1, 2, 3, 4)) IS TRUE
((1, 2, 3, 4) = (1, 2, 3, 5)) IS FALSE
((1, 2, 3, 4) < (1, 2, 3, 4)) IS FALSE
((1, 2, 3, 4) < (1, 2, 3, 5)) IS TRUE
((NULL, 1, NULL) = (NULL, 1, NULL)) IS UNKNOWN
((NULL, 1, NULL) = (NULL, 2, NULL)) IS FALSE
((NULL, 1, NULL) <> (NULL, 2, NULL)) IS TRUE
((NULL, 1, 2) <all operators> (NULL, 1, 2)) IS UNKNOWN
((1, NULL, ...) < (1, 2, ...)) IS UNKNOWN
((1, NULL, ...) < (2, NULL, ...)) IS TRUE
((2, NULL, ...) < (1, NULL, ...)) IS FALSE
```

BETWEEN

between predicate

```
<between predicate> ::= <row value predicand> <between predicate part 2>
```

```
<between predicate part 2> ::= [ NOT ] BETWEEN [ ASYMMETRIC | SYMMETRIC ] <row
value predicand> AND <row value predicand>
```

Specify a range comparison. The default is ASYMMETRIC. The expression X BETWEEN Y AND Z is equivalent to
(X >= Y AND X <= Z). Therefore if Y > Z, the BETWEEN expression is never true. The expression X BETWEEN
SYMMETRIC Y AND Z is equivalent to (X >= Y AND X <= Z) OR (X >= Z AND X <= Y). The
expression Z NOT BETWEEN ... is equivalent to NOT (Z BETWEEN ...). If any of the three <row value
predicand> evaluates to NULL, the result is UNKNOWN.

IN

in predicate

```
<in predicate> ::= <row value predicand> [ NOT ] IN <in predicate value>
```

```
<in predicate value> ::= <table subquery> | <left paren> <in value list> <right
paren>
```

```
| <left paren> UNNEST <left paren> <array value expression> <right paren> <right
paren>
```

```
<in value list> ::= <row value expression> [ { <comma> <row value expres-
sion> }... ]
```

Specify a quantified comparison. The expression X NOT IN Y is equivalent to NOT (X IN Y). The (<in
value list>) is converted into a table with one or more rows. The expression X IN Y is equivalent to X =
ANY Y, which is a <quantified comparison predicate>.

If the <table subquery> returns no rows, the result is FALSE. Otherwise the <row value predicand> is
compared one by one with each row of the <table subquery>.

If the comparison is TRUE for at least one row, the result is TRUE. If the comparison is FALSE for all rows, the result
is FALSE. Otherwise the result is UNKNOWN.

HyperSQL supports an extension to the SQL Standard to allow an array to be used in the <in predicate value>. This
is intended to be used with prepared statements where a variable length array of values can be used as the parameter
value for each call. The example below shows how this is used.

```
SELECT * from customer where firstname in ( UNNEST(?) )
```

LIKE

like predicate

```
<like predicate> ::= <character like predicate> | <octet like predicate>
```

```
<character like predicate> ::= <row value predicand> [ NOT ] LIKE <character
pattern> [ ESCAPE <escape character> ]
```

```
<character pattern> ::= <character value expression>
```

```
<escape character> ::= <character value expression>
```

```
<octet like predicate> ::= <row value predicand> [ NOT ] LIKE <octet pattern>
[ ESCAPE <escape octet> ]
```

```
<octet pattern> ::= <binary value expression>

<escape octet> ::= <binary value expression>
```

Specify a pattern-match comparison for character or binary strings. The `<row value predicand>` is always a `<string value expression>` of character or binary type. The `<character pattern>` or `<octet pattern>` is a `<string value expression>` in which the underscore and percent characters have special meanings. The underscore means match any one character, while the percent means match a sequence of zero or more characters. The `<escape character>` or `<escape octet>` is also a `<string value expression>` that evaluates to a string of exactly one character length. If the underscore or the percent is required as normal characters in the pattern, the specified `<escape character>` or `<escape octet>` can be used in the pattern before the underscore or the percent. The `<row value predicand>` is compared with the `<character pattern>` and the result of comparison is returned. If any of the expressions in the predicate evaluates to NULL, the result of the predicate is UNKNOWN. The expression `A NOT LIKE B` is equivalent to `NOT (A LIKE B)`. If the length of the escape is not 1 or it is used in the pattern not immediately before an underscore or a percent character, an exception is raised.

IS NULL

null predicate

```
<null predicate> ::= <row value predicand> IS [ NOT ] NULL
```

Specify a test for a null value. The expression `X IS NOT NULL` is NOT equivalent to `NOT (X IS NULL)`if the degree of the `<row value predicand>` is larger than 1. The rules are: If all fields are null, `X IS NULL` is TRUE and `X IS NOT NULL` is FALSE. If only some fields are null, both `X IS NULL` and `X IS NOT NULL` are FALSE. If all fields are not null, `X IS NULL` is FALSE and `X IS NOT NULL` is TRUE.

ALL and ANY

quantified comparison predicate

```
<quantified comparison predicate> ::= <row value predicand> <comp op> <quanti-
fier> <table subquery>

<quantifier> ::= <all> | <some>

<all> ::= ALL

<some> ::= SOME | ANY
```

Specify a quantified comparison. For a quantified comparison, the `<row value predicand>` is compared one by one with each row of the `<table sub query>`.

If the `<table subquery>` returns no rows, then if `ALL` is specified the result is TRUE, but if `SOME` or `ANY` is specified the result is FALSE.

If `ALL` is specified, if the comparison is TRUE for all rows, the result of the predicate is TRUE. If the comparison is FALSE for at least one row, the result is FALSE. Otherwise the result is UNKNOWN.

If `SOME` or `ANY` is specified, if the comparison is TRUE for at least one row, the result is TRUE. If the comparison is FALSE for all rows, the result is FALSE. Otherwise the result is UNKNOWN. Note that the IN predicate is equivalent to the SOME or ANY predicate using the `<equals operator>`.

In the examples below, the date of an invoice is compared to holidays in a given year. In the first example the invoice date must equal one of the holidays, in the second example it must be later than all holidays (later than the last holiday),

in the third example it must be on or after some holiday (on or after the first holiday), and in the fourth example, it must be before all holidays (before the first holiday).

```
invoice_date = SOME (SELECT holiday_date FROM holidays)
invoice_date > ALL (SELECT holiday_date FROM holidays)
invoice_date >= ANY (SELECT holiday_date FROM holidays)
invoice_date < ALL (SELECT holiday_date FROM holidays)
```

EXISTS

exists predicate

```
<exists predicate> ::= EXISTS <table subquery>
```

Specify a test for a non-empty set. If the evaluation of `<table subquery>` results in one or more rows, then the expression is TRUE, otherwise FALSE.

UNIQUE

unique predicate

```
<unique predicate> ::= UNIQUE <table subquery>
```

Specify a test for the absence of duplicate rows. The result of the test is either TRUE or FALSE (never UNKNOWN). The rows of the `<table subquery>` that contain one or more NULL values are not considered for this test. If the rest of the rows are distinct from each other, the result of the test is TRUE, otherwise it is FALSE. The distinctness of rows X and Y is tested with the predicate `X IS DISTINCT FROM Y`.

MATCH

match predicate

```
<match predicate> ::= <row value predicand> MATCH [ UNIQUE ] [ SIMPLE | PARTIAL
| FULL ] <table subquery>
```

Specify a test for matching rows. The default is MATCH SIMPLE without UNIQUE. The result of the test is either TRUE or FALSE (never UNKNOWN).

The interpretation of NULL values is different from other predicates and quite counter-intuitive. If the `<row value predicand>` is NULL, or all of its fields are NULL, the result is TRUE.

Otherwise, the `<row value predicand>` is compared with each row of the `<table subquery>`.

If SIMPLE is specified, if some field of `<row value predicate>` is NULL, the result is TRUE. Otherwise if `<row value predicate>` is equal to one or more rows of `<table subquery>` the result is TRUE if UNIQUE is not specified, or if UNIQUE is specified and only one row matches. Otherwise the result is FALSE.

If PARTIAL is specified, if the non-null values `<row value predicate>` are equal to those in one or more rows of `<table subquery>` the result is TRUE if UNIQUE is not specified, or if UNIQUE is specified and only one row matches. Otherwise the result is FALSE.

If FULL is specified, if some field of `<row value predicate>` is NULL, the result is FALSE. Otherwise if `<row value predicate>` is equal to one or more rows of `<table subquery>` the result is TRUE if UNIQUE is not specified, or if UNIQUE is specified and only one row matches.

Note that MATCH can also used be used in FOREIGN KEY constraint definitions. The exact meaning is described in the Schemas and Database Objects chapter.

OVERLAPS

overlaps predicate

```
<overlaps predicate> ::= <row value predicand> OVERLAPS <row value predicand>
```

Specify a test for an overlap between two datetime periods. Each `<row value predicand>` must have two fields and the fields together represent a datetime period. So the predicates is always in the form `(X1, X2) OVERLAPS (Y1, Y2)`. The first field is always a datetime value, while the second field is either a datetime value or an interval value.

If the second value is an interval value, it is replaced with the sum of the datetime value and itself, for example `(X1, X1 + X2) OVERLAPS (Y1, Y1 + Y 2)`.

If any of the values is NULL, the result is UNKNOWN.

The expression is true if there is there is any overlap between the two datetime periods. In the example below, the period is compared with a week long period ending yesterday.

```
(startdate, enddate) OVERLAPS (CURRENT_DATE - 7 DAY, CURRENT_DATE - 1 DAY)
```

IS DISTINCT

is distinct predicate

```
<distinct predicate> ::= <row value predicand> IS [ NOT ] DISTINCT FROM <row
value predicand>
```

Specify a test of whether two row values are distinct. The result of the test is either TRUE or FALSE (never UNKNOWN). The *degree* the two `<row value predicand>` must be the same. Each field of the first `<row value predicand>` is compared to the field of the second `<row value predicand>` at the same position. If one field is NULL and the other is not NULL, or if the elements are NOT equal, then the result of the expression is TRUE. If no comparison result is TRUE, then the result of the predicate is FALSE. The expression `X IS NOT DISTINCT FROM Y` is equivalent to `NOT (X IS DISTINCT FORM Y)`. The following check returns true if startdate is not equal to enddate. It also returns true if either startdate or enddate is NULL. It returns false in other cases.

```
startdate IS DISTINCT FROM enddate
```

Other Syntax Elements

search condition

search condition

```
<search condition> ::= <boolean value expression>
```

Specify a condition that is TRUE, FALSE, or UNKNOWN. A search condition is often a predicate.

PATH

path specification

```
<path specification> ::= PATH <schema name list>
```

```
<schema name list> ::= <schema name> [ { <comma> <schema name> }... ]
```

Specify an order for searching for a user-defined SQL-invoked routine. This is not currently supported by HyperSQL.

routine invocation

routine invocation

```
<routine invocation> ::= <routine name> <SQL argument list>

<routine name> ::= [ <schema name> <period> ] <qualified identifier>

<SQL argument list> ::= <left paren> [ <SQL argument> [ { <comma> <SQL argu-
ment> }... ] ] <right paren>

<SQL argument> ::= <value expression> | <target specification>
```

Invoke an SQL-invoked routine. Examples are given in the SQL-Invoked Routines chapter.

COLLATE

collate clause

```
<collate clause> ::= COLLATE <collation name>
```

Specify a default collation. This is not currently supported by HyperSQL

CONSTRAINT

constraint name definition

```
<constraint name definition> ::= CONSTRAINT <constraint name>

<constraint characteristics> ::= <constraint check time> [ [ NOT ] DEFERRABLE ]
| [ NOT ] DEFERRABLE [ <constraint check time> ]

<constraint check time> ::= INITIALLY DEFERRED | INITIALLY IMMEDIATE
```

Specify the name of a constraint and its characteristics. This is an optional element of CONSTRAINT definition, not yet supported by HyperSQL.

aggregate function

aggregate function

```
<aggregate function> ::= COUNT <left paren> <asterisk> <right paren> [ <filter
clause> ] | <general set function> [ <filter clause> ]

<general set function> ::= <set function type> <left paren> [ <set quantifier> ]
<value expression> <right paren>

<set function type> ::= <computational operation>

<computational operation> ::= AVG | MAX | MIN | SUM | EVERY | ANY | SOME | COUNT
| STDDEV_POP | STDDEV_SAMP | VAR_SAMP | VAR_POP

<set quantifier> ::= DISTINCT | ALL

<filter clause> ::= FILTER <left paren> WHERE <search condition> <right paren>
```

Specify a value computed from a collection of rows. An aggregate function is used exclusively in a `<query speci-fication>` and its use transforms a normal query into an aggregate query returning a single row instead of the group of multiple rows that the original query returns. For example, `SELECT acolumn <table expression>` is a query that returns the value of acolumn for all the rows the satisfy the given condition. But `SELECT MAX(acolumn) <table expression>` returns only one row, containing the largest value in that column. The query `SELECT COUNT(*) <table expression>` returns the count of rows, while `SELECT COUNT(acolumn) <table expression>` returns the count of rows where `acolumn IS NOT NULL`.

If the `<table expression>` is a grouped table, the aggregate function returns the result of the `COUNT` or `<com-putational operation>` for each group. In this case the result has the same number of rows as the original query. For example `SELECT SUM(acolumn) <table expression>` when `<table expression>` has a `GROUP BY` clause, returns the sum of values for `acolumn` in each group.

The AVG and SUM operations can be performed on numeric expressions only. AVG returns the average value, while SUM returns the sum of all non-null values. MAX and MIN return the minimum or the maximum value. If all values are NULL, the operations return NULL. The `COUNT(*)` operation returns the count of all values, while `COUNT(<value expression>)` returns the count of non-NULL values.

The EVERY, ANY and SOME operations can be performed on boolean expressions only. EVERY returns TRUE if all the values are TRUE, otherwise FALSE. ANY and SOME are the same operation and return TRUE if one of the values is TRUE, otherwise it returns FALSE.

The other operations perform the statistical functions STDDEV_POP, STDDEV_SAMP, VAR_SAMP, VAR_POP on numeric values. NULL values are ignored in calculations.

User defined aggregate functions can be defined and used instead of the built-in aggregate functions. Syntax and examples are given in the SQL-Invoked Routines chapter.

sort specification list

sort specification list

```
<sort specification list> ::= <value expression> [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Specify a sort order. A sort operation is performed on the result of a `<query expression>` or `<query speci-fication>` and sorts the result according to one or more `<value expression>`. The `<value expression>` is usually a single column of the result, but in some cases it can be a column of the `<table expression>` that is not used in the select list.

Data Access Statements

HyperSQL fully supports all of SQL-92 data access statements, plus some additions from SQL:2008. Due to time constraints, the current version of this Guide does not cover the subject fully. You are advised to consult an SQL book such as the recent O'Reilly title "SQL and Relational Theory" by C. J. Date.

Database queries are data access statements. The most commonly used data access statement is the SELECT statement, but there are other statements that perform a similar role. Data access statements access tables and return result tables. The returned result tables are falsely called result sets, as they are not necessarily sets of rows, but multisets of rows.

Result tables are formed by performing the following operations on base tables and views. These operations are loosely based on Relational Algebra.

JOIN operations

SET and *MULTISET* operations

SELECTION

PROJECTION

COMPUTING

COLUMN NAMING

GROUPING and *AGGREGATION*

SELECTION AFTER GROUPING OR AGGREGATION

SET and MULTISET (COLLECTION) OPERATIONS

ORDERING

SLICING

Conceptually, the operations are performed one by one in the above order if they apply to the given data access statement. In the example below a simple select statement is made more complex by adding various operations.

```
CREATE TABLE atable (a INT, b INT, c INT, d INT, e INT, f INT);
/* in the next SELECT, no join is performed and no further operation takes place */
SELECT * FROM atable
/* in the next SELECT, selection is performed by the WHERE clause, with no further action */
SELECT * FROM atable WHERE a + b = c
/* in the next SELECT, projection is performed after the other operations */
SELECT d, e, f FROM atable WHERE a + b = c
/* in the next SELECT, computation is performed after projection */
SELECT (d + e) / f FROM atable WHERE a + b = c
/* in the next two SELECT statements, column naming is performed in different ways*/
SELECT (a + e) / f AS calc, f AS div FROM atable WHERE a + b = c
SELECT dcol, ecol, fcol FROM atable(acol, bcol, ccol, dcol, ecol, fcol) WHERE acol + bcol = ccol
/* in the next SELECT, both grouping and aggregation is performed */
SELECT d, e, SUM(f) FROM atable GROUP BY d, e
/* in the next SELECT, selection is performed after grouping and aggregation is performed */
SELECT d, e, SUM(f) FROM atable GROUP BY d, e HAVING SUM(f) > 10
/* in the next SELECT, a UNION is performed on two selects from the same table */
SELECT d, e, f FROM atable WHERE d = 3 UNION SELECT a, b, c FROM atable WHERE a = 30
/* in the next SELECT, ordering is performed */
SELECT (a + e) / f AS calc, f AS div FROM atable WHERE a + b = c ORDER BY calc DESC, div NULLS
 LAST
/* in the next SELECT, slicing is performed after ordering */
SELECT * FROM atable WHERE a + b = c ORDER BY a FETCH 5 ROWS ONLY
```

The next sections discuss various types of tables and operations involved in data access statements.

Table

In data access statements, a table can be a database table (or view) or an ephemeral table formed for the duration of the query. Some types of table are <table primary> and can participate in joins without the use of extra parentheses. The BNF in the Table Primary section below lists different types of <table primary>:

Tables can also be formed by specifying the values that are contained in them:

```
<table value constructor> ::= VALUES <row value expression list>
```

```
<row value expression list> ::= <table row value expression> [ { <comma> <table
row value expression> }... ]
```

In the example below a table with two rows and 3 columns is constructed out of some values:

```
VALUES (12, 14, null), (10, 11, CURRENT_DATE)
```

When a table is used directly in a UNION or similar operation, the keyword TABLE is used with the name:

```
<explicit table> ::= TABLE <table or query name>
```

In the examples below, all rows of the two tables are included in the union. The keyword TABLE is used in the first example. The two examples below are equivalent.

```
TABLE atable UNION TABLE anothertable
SELECT * FROM atable UNION SELECT * FROM anothertable
```

Query Specification

A query specification is a SELECT statement. It is the most common form of `<derived table>`. A `<table expression>` is a base table, a view or any form of allowed derived table. The SELECT statement performs projection, naming, computing or aggregation on the rows of the `<table expression>`.

```
<query specification> ::= SELECT [ DISTINCT | ALL ] <select list> <table ex-
pression>
```

```
<select list> ::= <asterisk> | <select sublist> [ { <comma> <select sub-
list> }... ]
```

```
<select sublist> ::= <derived column> | <qualified asterisk>
```

```
<qualified asterisk> ::= <asterisked identifier chain> <period> <asterisk>
```

```
<asterisked identifier chain> ::= <asterisked identifier> [ { <period> <aster-
isked identifier> }... ]
```

```
<asterisked identifier> ::= <identifier>
```

```
<derived column> ::= <value expression> [ <as clause> ]
```

```
<as clause> ::= [ AS ] <column name>
```

The qualifier DISTINCT or ALL apply to the results of the SELECT statement after all other operations have been performed. ALL simply returns the rows, while DISTINCT compares the rows and removes the duplicate ones.

Projection is performed by the `<select list>`.

A single `<asterisk>` means all columns of the `<table expression>` are included, in the same order as they appear in the `<table expression>`. An asterisk qualified by a table name means all the columns of the qualifier table name are included.

A derived column is a `<value expression>`, optionally named with the `<as clause>`. A `<value expression>` can be many things. Common types include: the name of a column in the `<table expression>`; an expression based on different columns or constant values; a function call; an aggregate function; a CASE WHEN expression.

Table Expression

A table expression is part of the SELECT statement and consists of the FROM clause with optional other clauses that performs selection (of rows) and grouping from the table(s) in the FROM clause.

```
<table expression> ::= <from clause> [ <where clause> ] [ <group by clause> ]
[ <having clause> ]

<from clause> ::= FROM <table reference> [ { <comma> <table reference> }... ]

<table reference> ::= <table primary> | <joined table>

<table primary> ::= <table or query name> [ [ AS ] <correlation name> [ <left
paren> <derived column list> <right paren> ] ]

| <derived table> [ AS ] <correlation name> [ <left paren> <derived column
list> <right paren> ]

| <lateral derived table> [ AS ] <correlation name> [ <left paren> <derived
column list> <right paren> ]

| <collection derived table> [ AS ] <correlation name> [ <left paren> <derived
column list> <right paren> ]

| <table function derived table> [ AS ] <correlation name> [ <left paren>
<derived column list> <right paren> ]

| <parenthesized joined table>

<where clause> ::= WHERE <boolean value expression>

<group by clause> ::= GROUP BY [ <set quantifier> ] <grouping element> [ { <comma>
<grouping element> }... ]

<having clause> ::= HAVING <boolean value expression>
```

The <from clause> contains one or more <table reference> separated by commas. A table reference is often a table or view name or a joined table.

The <where clause> filters the rows of the table in the <from clause> and removes the rows for which the search condition is not TRUE.

The <group by clause> is a comma separated list of columns of the table in the <from clause> or expressions based on the columns.

When a <group by clause> is used, only the columns used in the <group by clause> or expressions used there, can be used in the <select list>, together with any <aggregate function> on other columns. A <group by clause> compares the rows and groups together the rows that have the same values in the columns of the <group by clause>. Then any <aggregate function> in the <select list> is performed on each group, and for each group, a row is formed that contains the values of the columns of the <group by clause> and the values returned from each <aggregate function>. In the first example below, a simple column reference is used in GROUP BY, while in the second example, an expression is used.

```
CREATE TABLE atable (a INT, b INT, c INT, d INT, e INT, f INT);
SELECT d, e, f FROM atable WHERE a + b = c GROUP BY d, e, f
SELECT d + e, SUM(f) FROM atable WHERE a + b = c GROUP BY d + e HAVING SUM(f) > 2 AND d + e > 4
```

A <having clause> filters the rows of the table that is formed after applying the <group by clause> using its search condition. The search condition must be an expression based on the expressions in the GROUP BY list or the aggregate functions used.

Table Primary

Table primary refers to different forms of table reference in the FROM clause.

The simplest form of reference is simply a name. This is the name of a table, a view, a transition table in a trigger definition, or a query name specified in the WITH clause of a query expression.

```
<table or query name> ::= <table name> | <transition table name> | <query name>
```

A query expression that is enclosed in parentheses and returns from zero to many rows is a `<table subquery>`. In a `<derived table>` the query expression is self contained and cannot reference the columns of other table references.

```
<derived table> ::= <table subquery>
```

When the word LITERAL is used before a `<table subquery>`, it means the query expression can reference the columns of other table references that precede it.

```
<lateral derived table> ::= LATERAL <table subquery>
```

UNNEST is similar to LATERAL, but instead of a query expression, and expression that returns an array is used. This expression is converted into a table which has one column that contains the elements of the array, and, if WITH ORDINALITY is used, a second column that contains the index of each element. The array expression usually contains a reference to a column of the table reference preceding the current table reference.

```
<collection derived table> ::= UNNEST <left paren> <array value expression>
<right paren> [ WITH ORDINALITY ]
```

When TABLE is used in this context, it also converts an array value expression to a table, but this array must be the result of a function call. A function that returns a MULTISET can also be used in this context and each row of the multiset is expanded into a row of the table.

```
<table function derived table> ::= TABLE <left paren> <collection value expres-
sion> <right paren>
```

The column list that is specified for the table reference must contain names that are unique within the list

```
<derived column list> ::= <column name list>
```

```
<column name list> ::= <column name> [ { <comma> <column name> }... ]
```

A parenthesized joined table is simply a joined table contained in parentheses. Joined tables are discussed below.

```
<parenthesized joined table> ::= <left paren> <parenthesized joined table>
<right paren> | <left paren> <joined table> <right paren>
```

Joined Table

Joins are operators with two table as the operands, resulting in a third table, called joined table. All join operators are evaluated left to right, therefore, with multiple joins, the table resulting from the first join operator becomes an operand of the next join operator. Parentheses can be used to group sequences of joined tables and change the evaluation order. So if more than two tables are joined together with join operators, the end result is also a joined table. There are different types of join, each producing the result table in a different way.

CROSS JOIN

The simplest form of join is CROSS JOIN. The CROSS JOIN of two tables is a table that has all the columns of the first table, followed by all the columns of the second table, in the original order. Each row of the first table is combined with each row of the second table to fill the rows of the new table. If the rows of each table form a set, then the rows of the CROSS JOIN table form the Cartesian product of the rows of the two table operands.

The CROSS JOIN can be expressed in two forms. The first form is `A CROSS JOIN B`. The second form is `A, B`. This type of join is not generally very useful, as it returns large result tables.

UNION JOIN

The UNION JOIN has limited use in queries. The result table has the same columns as that of CROSS JOIN. Each row of the first table is extended to the right with nulls and added to the new table. Each row of the second table is extended to the left with nulls and added to the new table. The UNION JOIN is expressed as `A UNION JOIN B`. This should not be confused with `A UNION B`, which is a set operation. Union join is for special applications and is not commonly used.

JOIN ... ON

The condition join is similar to CROSS JOIN, but a condition is tested for each row of the new table and the row is created only if the condition is true. This form of join is expressed as `A JOIN B ON (<search condition>)`.

Equijoin is a condition join in which the search condition is an equality condition between on or more pairs of columns from the two table. Equijoin is the most commonly used type of join.

JOIN ... USING

NATURAL JOIN

Joins with USING or NATURAL keywords joins are similar to an equijoin but they cannot be replaced simply with an equijoin. The new table is formed with the specified or implied shared columns of the two tables, followed by the rest of the columns from each table. In NATURAL JOIN, the shared columns are all the column pairs that have the same name in the first and second table. In JOIN USING, only columns names that are specified by the USING clause are shared. The joins are expressed as `A NATURAL JOIN B`, and `A JOIN B USING (<comma separated column name list>)`.

The columns of the joined table are formed by the following procedures: In JOIN ... USING the shared columns are added to the joined table in the same order as they appear in the column name list. In NATURAL JOIN the shared columns are added to the joined table in the same order as they appear in the first table. In bother forms of join, the non-shared columns of the first table are added in the order they appear in the first table, finally the non-shared columns of the second table are added in the order they appear in the second table.

The type of each shared column of the joined table is based on the type of the columns in the original tables. If the original types are not exactly the same, the type of the shared column is formed by type aggregation. Type aggregations selects a type that can represent values of both aggregated types. Simple type aggregation picks one of the types. For example SMALLINT and INTEGER, results in INTEGER, or VARCHAR(10) and VARCHAR(20) results in VARCHAR(20). More complex type aggregation inherits properties from both types. For example DECIMAL(8) and DECIMAL (6,2) results in DECIMAL (8,2).

OUTER JOIN

LEFT, RIGHT and FULL OUTER JOIN

The three qualifiers can be added to all types of JOIN apart from CROSS JOIN and UNION JOIN. First the new table is populated with the rows from the original join. If LEFT is specified, all the rows from the first table that did not make it into the new table are extended to the right with nulls and added to the table. If RIGHT is specified, all the rows from the second table that did not make it into the new table are extended to the left with nulls and added to the

table. If FULL is specified, the addition of leftover rows is performed from both the first and the second table. These forms are expressed by prefixing the join specification with the given keyword. For example `A LEFT OUTER JOIN B ON (<search condition>)` or `A NATURAL FULL OUTER JOIN B` or `A FULL OUTER JOIN B USING (<comma separated column name list>)`.

Selection

Despite the name, selection has nothing to do with the list of columns in a SELECT statement. In fact, it refers to the search condition used to limit the rows that from a result table (selection of rows, not columns). In SQL, simple selection is expressed with a WHERE condition appended to a single table or a joined table. In some cases, this method of selection is the only method available. But when it is possible to perform the selection with join conditions, this is the better method, as it results in a clearer expression of the query.

Projection

Projection is selection of the columns from a simple or joined table to form a result table. Explicit projection is performed in the SELECT statement by specifying the select column list. Some form of projection is also performed in JOIN ... USING and NATURAL JOIN.

The joined table has columns that are formed according to the rules mentioned above. But in many cases, not all the columns are necessary for the intended operation. If the statement is in the form, SELECT * FROM <joined table>, then all the columns of <joined table> are returned. But normally, the columns to be returned are specified after the SELECT keyword, separated from each other with commas.

Computed Columns

In the select list, it is possible to use expressions that reference any columns of <joined table>. Each of these expressions forms a computed column. It is computed for each row of the result table, using the values of the columns of the <joined table> for that row.

Naming

Naming is used to hide the original names of tables or table columns and to replace them with new names in the scope of the query. Naming is also used for defining names for computed columns.

Naming in Joined Table

Naming is performed by adding a new name after a table's real name and by adding a list of column names after the new table name. Both table naming and column naming are optional, but table naming is required for column naming. The expression `A [AS] X (<comma separated column name list>)` means table A is used in the query expression as table X and its columns are named as in the given list. The original name A, or its original column names, are not visible in the scope of the query. The BNF is given below. The `<correlation name>` can be the same or different from the name of the table. The `<derived column list>` is a comma separated list of column names. The degree of this list must be equal to the degree of the table. The column names in the list must be distinct. They can be the same or different from the names of the table's columns.

```
<table or query name> [ [ AS ] <correlation name> [ <left paren> <derived column
list> <right paren> ] ]
```

In the examples below, the columns of the original tables are named (a, b, c, d, e, f). The two queries are equivalent. In the second query, the table and its columns are renamed and the new names are used in the WHERE clauses:

```
CREATE TABLE atable (a INT, b INT, c INT, d INT, e INT, f INT);
SELECT d, e, f FROM atable WHERE a + b = c
```

```
SELECT x, y, z FROM atable AS t (u, v, w, x, y, z)  WHERE u + v = w
```

Naming in Select List

Naming in the SELECT list logically takes place after naming in the joined table. The new names for columns are not visible in the immediate query expression or query expression. They become visible in the ORDER BY clause and in the result table that is returned to the user. Or if the query expression is used as a derived table in an enclosing query expression.

In the example below, the query is on the same table but with column renaming in the Select list. The new names are used in the ORDER BY clause:

```
SELECT x + y AS xysum, y + z AS yzsum FROM atable AS t (u, v, w, x, y, z)  WHERE u + v = w ORDER
 BY xysum, yzsum
```

If the names `xysum` or `yzsum` are not used, the computed columns cannot be referenced in the ORDER BY list.

Name Resolution

In a joined table, if a column name appears in tables on both sides then any reference to the name must use the table name in order to specify which table is being referred to.

Grouping Operations

Grouping Operations

Grouping results in the elimination of duplicate rows. A grouping operation is performed after the operations discussed above. A simple form of grouping is performed by the use of DISTINCT after SELECT. This eliminates all the duplicate rows (rows that have the same value in each of their columns when compared to another row). The other form of grouping is performed with the GROUP BY clause. This form is usually used together with aggregation.

Aggregation

Aggregation is an operation that computes a single value from the values of a column over several rows. The operation is performed with an aggregate function. The simplest form of aggregation is counting, performed by the COUNT function.

Other common aggregate functions return the maximum, minimum and average value among the values in different rows.

Set Operations

Set and Multiset Operations

While join operations generally result in laterally expanded tables, SET and COLLECTION operations are performed on two tables that have the same degree and result in a table of the same degree. The SET operations are UNION, INTERSECT and EXCEPT (difference). When each of these operations is performed on two tables, the collection of rows in each table and in the result is reduced to a set of rows, by eliminating duplicates. The set operations are then performed on the two tables, resulting in the new table which itself is a set of rows. Collection operations are similar but the tables are not reduced to sets before or after the operation and the result is not necessarily a set, but a collection of rows.

The set operations on two tables A and B are: A `UNION [DISTINCT]` B, A `INTERSECT [DISTINCT]` B and A `EXCEPT [DISTINCT]` B. The result table is formed in the following way: The UNION operation adds all the rows from A and B into the new table, but avoids copying duplicate rows. The INTERSECT operation copies only those

rows from each table that also exist in the other table, but avoids copying duplicate rows. The EXCEPT operation copies those rows from the first table which do not exist in the second table, but avoids copying duplicate rows.

The collection operations are similar to the set operations, but can return duplicate rows. They are A UNION ALL B, A INTERSECT ALL B and A EXCEPT ALL B. The UNION ALL operation adds all the rows from A and B into the new table. The INTERSECT operation copies only those rows from each table that also exist in the other table. If n copies of a rows exists in one table, and m copies in the other table, the number of copies in the result table is the smaller of n and m. The EXCEPT operation copies those rows from the first table which do not exist in the second table. If n copies of a row exist in the first table and m copies in the second table the number of copies in the result table is n-m, or if n < m, then zero.

Query Expression

A query expression consists of an optional WITH clause and a query expression body. The WITH clause lists one or more named ephemeral tables that can be referenced in the query expression body.

```
<query expression> ::= [ <with clause> ] <query expression body>

<with clause> ::= WITH <with list>

<with list> ::= <with list element> [ { <comma> <with list element> }... ]

<with list element> ::= <query name> [ <left paren> <with column list> <right
paren> ] AS <left paren> <query expression> <right paren>

<with column list> ::= <column name list>
```

A query expression body refers to a table formed by using UNION and other set operations. The query expression body is evaluated from left to right and the INTERSECT operator has precedence over the UNION and EXCEPT operators. A simplified BNF is given below:

```
<query expression body> ::= <query term> | <query expression body> UNION |
EXCEPT [ ALL | DISTINCT ] [ <corresponding spec> ] <query term>

<query term> ::= <query primary> | <query term> INTERSECT [ ALL | DISTINCT ]
[ <corresponding spec> ] <query term>

<query primary> ::= <simple table> | <left paren> <query expression body> [ <order
by clause> ] [ <result offset clause> ] [ <fetch first clause> ] <right paren>

<simple table> ::= <query specification> | <table value constructor> | <explicit
table> <explicit table> ::= TABLE <table or query name>

<corresponding spec> ::= CORRESPONDING [ BY <left paren> <column name list>
<right paren> ]
```

A <query term> and a <query primary> can be a SELECT statement, an <explicit table>, or a <table value constructor>.

The CORRESPONDING clause is optional. If it is not specified, then the <query term> and the <query primary> must have the same number of columns. If CORRESPONDING is specified, the two sides need not have the same number of columns. If no column list is used with CORRESPONDING, then all the column names that are common in the tables on two sides are used in the order in which they appear in the first table. If a columns list is used, it allows you to select only some columns of the tables on the left and right side to create the new table. In the example below the columns named u and v from the two SELECT statements are used to create the UNION table.

```
SELECT * FROM atable UNION CORRESPONDING BY (u, v) SELECT * FROM anothertable
```

The type of each column of the query expression is determined by combining the types of the corresponding columns from the two participating tables.

Ordering

When the rows of the result table have been formed, it is possible to specify the order in which they are returned to the user. The ORDER BY clause is used to specify the columns used for ordering, and whether ascending or descending ordering is used. It can also specify whether NULL values are returned first or last.

```
SELECT x + y AS xysum, y + z AS yzsum FROM atable AS t (u, v, w, x, y, z)  WHERE u + v = w ORDER
 BY xysum NULLS LAST, yzsum NULLS FIRST
```

The ORDER BY clause specifies one or more `<value expressions>`. The list of rows is sorted according to the first `<value expression>`. When some rows are sorted equal then they are sorted according to the next `<value expression>` and so on.

```
<order by clause> ::= ORDER BY <sort specification> [ { <comma> <sort speci-
fication> }... ]
```

```
<sort specification> ::= <value expression> [ ASC | DESC ] [ NULLS FIRST |
NULLS LAST ]
```

Slicing

A different form of limiting the rows can be performed on the result table after it has been formed according to all the other operations (selection, grouping, ordering etc.). This is specified by the FETCH ... ROWS and OFFSET clauses of a SELECT statement. In this form, the specified OFFSET rows are removed from start of the table, then up to the specified FETCH rows are kept and the rest of the rows are discarded.

```
<result offset clause> ::= OFFSET <offset row count> { ROW | ROWS }
```

```
<fetch first clause> ::= FETCH { FIRST | NEXT } [ <fetch first row count> ]
{ ROW | ROWS } ONLY
```

```
<limit clause> ::= LIMIT [ <fetch first row count> ]
```

A slicing operation takes the result set that has been already processed and ordered. It then discards the specified number of rows from the start of the result set and returns the specified number of rows after the discarded rows.

```
SELECT a, b FROM atable WHERE d < 5 ORDER BY absum OFFSET 3 FETCH 2 ROWS ONLY
SELECT a, b FROM atable WHERE d < 5 ORDER BY absum OFFSET 3 LIMIT 2 /* alternative keyword */
```

Data Change Statements

Delete Statement

DELETE FROM

delete statement: searched

```
<delete statement: searched> ::= DELETE FROM <target table> [ [ AS ] <correlation
name> ] [ WHERE <search condition> ]
```

Delete rows of a table. The search condition is a `<boolean value expression>` that is evaluated for each row of the table. If the condition is true, the row is deleted. If the condition is not specified, all the rows of the table are

deleted. In fact, an implicit SELECT is performed in the form of `SELECT * FROM <target table> [ WHERE <search condition>]` and the selected rows are deleted. When used in JDBC, the number of rows returned by the implicit SELECT is returned as the update count.

If there are FOREIGN KEY constraints on other tables that reference the subject table, and the FOREIGN KEY constraints have referential actions, then rows from those other tables that reference the deleted rows are either deleted, or updated, according to the specified referential actions.

In the second example below the rows that have the maximum value for column A are deleted;

```
DELETE FROM T WHERE C > 5
DELETE FROM T AS TT WHERE TT.A = (SELECT MAX(A) FROM T)
```

Truncate Statement

TRUNCATE TABLE

truncate table statement

```
<truncate table statement> ::= TRUNCATE TABLE <target table> [ <identity column
restart option> ]

<identity column restart option> ::= CONTINUE IDENTITY | RESTART IDENTITY
```

Delete all rows of a table without firing its triggers. This statement can only be used on base tables (not views). If the table is referenced in a FOREIGN KEY constraint, the statement causes an exception. Triggers defined on the table are not executed with this statement. The default for `<identity column restart option>` is `CONTINUE IDENTITY`. This means no change to the IDENTITY sequence of the table. If `RESTART IDENTITY` is specified, then the sequence is reset to its start value.

TRUNCATE is faster than ordinary DELETE. The TRUNCATE statement is an SQL Standard data change statement, therefore it is performed under transaction control and can be rolled back if the connection is not in the auto-commit mode.

Insert Statement

INSERT INTO

insert statement

```
<insert statement> ::= INSERT INTO <target table> <insert columns and source>

<insert columns and source> ::= <from subquery> | <from constructor> | <from
default>

<from subquery> ::= [ <left paren> <insert column list> <right paren> ] [ <over-
ride clause> ] <query expression>

<from constructor> ::= [ <left paren> <insert column list> <right paren> ]
[ <override clause> ] <contextually typed table value constructor>

<override clause> ::= OVERRIDING USER VALUE | OVERRIDING SYSTEM VALUE

<from default> ::= DEFAULT VALUES

<insert column list> ::= <column name list>
```

Insert new rows in a table. An INSERT statement inserts one or more rows into the table.

The special form, `INSERT INTO <target table> DEFAULT VALUES` can be used with tables which have a default value for each column.

With the other forms of INSERT, the optional `(<insert column list>)` specifies to which columns of the table the new values are assigned.

In one form, the inserted values are from a `<query expression>` and all the rows that are returned by the `<query expression>` are inserted into the table. If the `<query expression>` returns no rows, nothing is inserted.

In the other form, a comma separated list of values called `<contextually typed table value construc-tor>` is used to insert one or more rows into the table. This list is contextually typed, because the keywords NULL and DEFAULT can be used for the values that are assigned to each column of the table. The keyword DEFAULT means the default value of the column and can be used only if the target column has a default value or is an IDENTITY or GENERATED column of the table.

The `<override clause>` must be used when a value is explicitly assigned to a column that has been defined as GENERATED ALWAYS AS IDENTITY. The clause, OVERRIDE SYSTEM VALUE means the provided values are used for the insert, while OVERRIDING USER VALUE means the provided values are simply ignored and the values generated by the system are used instead.

An array can be inserted into a column of the array type by using literals, by specifying a parameter in a prepared statement or an existing array returned by query expression. The last example below inserts an array.

The rows that are inserted into the table are checked against all the constraints that have been declared on the table. The whole INSERT operation fails if any row fails to inserted due to constraint violation. Examples:

```
INSERT INTO T DEFAULT VALUES /* all columns of T have DEFAULT clauses */
INSERT INTO T (SELECT * FROM Z) /* table Z has the same columns as table T */
INSERT INTO T (A,B) VALUES ((1,2),(3,NULL), (DEFAULT,6)) /* three rows are inserted into table T
 */
INSERT INTO T VALUES 3, ARRAY['hot','cold']
```

Update Statement

UPDATE

update statement: searched

```
<update statement: searched> ::= UPDATE <target table> [ [ AS ] <correlation
name> ] SET <set clause list> [ WHERE <search condition> ]
```

Update rows of a table. An UPDATE statement selects rows from the `<target table>` using an implicit SELECT statement formed in the following manner:

```
SELECT * FROM <target table> [ [ AS ] <correlation name> ] [ WHERE <search
condition> ]
```

Then it applies the `SET <set clause list>` expression to each selected row.

If the implicit SELECT returns no rows, no update takes place. When used in JDBC, the number of rows returned by the implicit SELECT is returned as the update count.

If there are FOREIGN KEY constraints on other tables that reference the subject table, and the FOREIGN KEY constraints have referential actions, then rows from those other tables that reference the updated rows are updated, according to the specified referential actions.

The rows that are updated are checked against all the constraints that have been declared on the table. The whole UPDATE operation fails if any row violates any constraint.

set clause list

set clause list

```
<set clause list> ::= <set clause> [ { <comma> <set clause> }... ]

<set clause> ::= <multiple column assignment> | <set target> <equals operator>
<update source>

<multiple column assignment> ::= <set target list> <equals operator> <assigned
row>

<set target list> ::= <left paren> <set target> [ { <comma> <set target> }... ]
<right paren>

<assigned row> ::= <contextually typed row value expression>

<set target> ::= <column name>

<update source> ::= <value expression> | <contextually typed value specification>
```

Specify a list of assignments. This is used in UPDATE, MERGE and SET statements to assign values to a scalar or row target.

Apart from setting a whole target to a value, a SET statement can set individual elements of an array to new values. The last example below shows this form of assignment to the array in the column named B.

In the examples given below, UPDATE statements with single and multiple assignments are shown. Note in the third example, a SELECT statement is used to provide the update values for columns A and C, while the update value for column B is given separately. The SELECT statement must return exactly one row . In this example the SELECT statement refers to the existing value for column C in its search condition.

```
UPDATE T SET A = 5 WHERE ...
UPDATE T SET (A, B) = (1, NULL) WHERE ...
UPDATE T SET (A, C) = (SELECT X, Y FROM U WHERE Z = C), B = 10 WHERE ...
UPDATE T SET A = 3, B[3] = 'warm'
```

Merge Statement

MERGE INTO

merge statement

```
<merge statement> ::= MERGE INTO <target table> [ [ AS ] <merge correlation
name> ] USING <table reference> ON <search condition> <merge operation speci-
fication>

<merge correlation name> ::= <correlation name>

<merge operation specification> ::= <merge when clause>...

<merge when clause> ::= <merge when matched clause> | <merge when not matched
clause>

<merge when matched clause> ::= WHEN MATCHED THEN <merge update specification>
```

```
<merge when not matched clause> ::= WHEN NOT MATCHED THEN <merge insert spec-
ification>
```

```
<merge update specification> ::= UPDATE SET <set clause list>
```

```
<merge insert specification> ::= INSERT [ <left paren> <insert column list>
<right paren> ] [ <override clause> ] VALUES <merge insert value list>
```

```
<merge insert value list> ::= <left paren> <merge insert value element> [ { <com-
ma> <merge insert value element> }... ] <right paren>
```

```
<merge insert value element> ::= <value expression> | <contextually typed value
specification>
```

Update rows, or insert new rows into the `<target table>`. The MERGE statement uses a second table, specified by `<table reference>`, to determine the rows to be updated or inserted. It is possible to use the statement only to update rows or to insert rows, but usually both update and insert are specified.

The `<search condition>` matches each row of the `<table reference>` with each row of the `<target table>`. If the two rows match then the UPDATE clause is used to update the matching row of the target table. Those rows of `<table reference>` that have no matching rows are then used to insert new rows into the `<target table>`. Therefore, a MERGE statement can update between 0 and all the rows of the `<target table>` and can insert between 0 and the number of the rows in `<table reference>` into the `<target table>`. If any row in the `<target table>` matches more than one row in `<table reference>` a cardinality error is raised. On the other hand, several rows in the `<target table>` can matches a single row in `<table reference>` without any error. The constraints and referential actions specified on the database tables are enforced the same way as for an update and an insert statement.

The MERGE statement can be used with only the WHEN NOT MATCHED clause as a conditional INSERT statement that inserts a row if no existing rows match a condition.

In the first example below, the table originally contains two rows for different furniture. The `<table reference>` is the `(VALUES(1, 'conference table'), (14, 'sofa'), (5, 'coffee table'))` expression, which evaluates to a table with 3 rows. When the x value for a row matches an existing row, then the existing row is updated. When the x value does not match, the row is inserted. Therefore one row of table t is updated from 'dining table' to 'conference table', and two rows are inserted into table t. The second example uses a SELECT statement as the source of the values for the MERGE.

In the third example, a new row in inserted into the table only when the primary key for the new row does not exist. This example uses parameters and should be executed as a JDBC PreparedStatement.

```
CREATE TABLE t (id INT PRIMARY KEY, description VARCHAR(100))
INSERT INTO t VALUES (1, 'dining table'), (2, 'deck chair')
MERGE INTO t USING (VALUES(1, 'conference table'), (14, 'sofa'), (5, 'coffee table'))
   AS vals(x,y) ON t.id = vals.x
   WHEN MATCHED THEN UPDATE SET t.description = vals.y
   WHEN NOT MATCHED THEN INSERT VALUES vals.x, vals.y

MERGE INTO t USING (SELECT * FROM tt WHERE acol = 2) AS vals(x,y) ON t.id = vals.x
   WHEN MATCHED THEN UPDATE SET t.description = vals.y
   WHEN NOT MATCHED THEN INSERT VALUES vals.x, vals.y

MERGE INTO t USING (VALUES(CAST(? AS INT))) AS vals(x) ON t.id = vals.x
   WHEN NOT MATCHED THEN INSERT VALUES vals.x, ?
```

Chapter 8. SQL-Invoked Routines

Fred Toussi, The HSQL Development Group

$Revision: 3643 $

Copyright 2010 Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2010-06-06 23:04:17 -0400 (Sun, 06 Jun 2010) $

SQL-invoked routines are functions and procedures called from SQL. HyperSQL 2.0 supports routines conforming to two parts of the SQL Standard. Routines written in the SQL language are supported in conformance to SQL/PSM (Persistent Stored Modules) specification. Routines written in Java are supported in (loose) conformance to SQL/JRT specification. In addition, HyperSQL's previous non-standard support for calling Java routines without prior method definition is retained and enhanced in the latest version by extending the SQL/JRT specification.

HyperSQL also supports user defined aggregate functions written in the SQL language. This feature is an extension to the SQL Standard.

SQL-invoked routines are schema objects. Naming and referencing follows conventions common to all schema objects. The same routine name can be defined in two different schemas and used with schema-qualified references.

A routine is either a procedure or a function.

A function:

• is defined with CREATE FUNCTION

• always returns a value

• does not modify the data in the database

• is called as part of an SQL statement

• can have parameters

• can be polymorphic

A procedure:

• is defined with CREATE PROCEDURE

• can return one or more values

• can modify the data in the database

• is called separately, using the CALL statement

• can have parameters

• can be polymorphic

Definition of routine signature and characteristics, name resolution and invocation are all implemented uniformly for routines written in SQL or Java.

SQL Language Routines (PSM)

The PSM (Persistent Stored Module) specification extends the SQL language to allow definition of both SQL Function and SQL procedure bodies with the same structure and the same control statements (such as conditional and loop statements) with minor exceptions.

The routine body is a SQL statement. In its simplest form, the body is a single SQL statement. A simple example of a function is given below:

```
CREATE FUNCTION an_hour_before (t TIMESTAMP)
  RETURNS TIMESTAMP
  RETURN t - 1 HOUR
```

An example of the use of the function in an SQL statement is given below:

```
SELECT an_hour_before(event_timestamp) AS notification_timestamp, event_name FROM events;
```

A simple example of a procedure is given below:

```
CREATE PROCEDURE new_customer(firstname VARCHAR(50), lastname VARCHAR(50))
  MODIFIES SQL DATA
  INSERT INTO CUSTOMERS VALUES (DEFAULT, firstname, lastname, CURRENT_TIMESTAMP)
```

The procedure inserts a row into an existing table with the definition given below:

```
CREATE TABLE customers(id INTEGER GENERATED BY DEFAULT AS IDENTITY, firstname VARCHAR(50),
 lastname VARCHAR(50), added TIMESTAMP);
```

An example of the use of the procedure is given below:

```
CALL new_customer('JOHN', 'SMITH');
```

The routine body is often a compound statement. A compound statement can contain one or more SQL statements, which can include control statements, as well as nested compound statements.

Routine Statements

The following SQL Statements can be used only in routines.

```
<handler declaration>
```

```
<assignment statement>
```

```
<compound statement>
```

```
<case statement>
```

```
<if statement>
```

```
<while statement>
```

```
<repeat statement>
```

```
<for statement>
```

```
<loop statement>
```

```
<iterate statement
```

```
<leave statement>
```

```
<signal statement>
```

```
<resignal statement>
```

```
<return statement>
```

```
<select statement: single row>
```

The following SQL Statements can be used in procedures but not in functions.

```
<call statement>
```

```
<delete statement>
```

```
<insert statement>
```

```
<update statement>
```

```
<merge statement>
```

As shown in the examples below, the formal parameters and the variables of the routine can be used in statements, similar to the way a column reference is used.

Compound Statement

A compound statement is enclosed in a BEGIN / END block with optional labels. It can contain one or more `<SQL variable declaration>` or `<handler declaration>` before at least one SQL statement. The BNF is given below:

```
<compound statement> ::= [ <beginning label> <colon> ] BEGIN [[NOT] ATOMIC]
[{<SQL variable declaration> <semicolon>} ...] [{<handler declaration> <semi-
colon>}...] {<SQL procedure statement> <semicolon>} ... END [ <ending label> ]
```

An example of a simple compound statement body is given below. It performs the common task of inserting related data into two table. The IDENTITY value that is automatically inserted in the first table is retrieved using the IDENTITY() function and inserted into the second table.

```
CREATE PROCEDURE new_customer(firstname VARCHAR(50), lastname  VARCHAR(50), address VARCHAR(100))
  MODIFIES SQL DATA
    BEGIN ATOMIC
    INSERT INTO customers VALUES (DEFAULT, firstname, lastname, CURRENT_TIMESTAMP);
    INSERT INTO addresses VALUES (DEFAULT, IDENTITY(), address);
  END
```

Variables

A `<variable declaration>` defines the name and data type of the variable and, optionally, its default value. In the next example, a variable is used to hold the IDENTITY value. In addition, the formal parameters of the procedure are identified as input parameters with the use of the optional IN keyword. This procedure does exactly the same job as the procedure in the previous example.

```
CREATE PROCEDURE new_customer(IN firstname VARCHAR(50), IN lastname VARCHAR(50), IN address
 VARCHAR(100))
  MODIFIES SQL DATA
  BEGIN ATOMIC
    DECLARE temp_id INTEGER;
```

```
   INSERT INTO CUSTOMERS VALUES (DEFAULT, firstname, lastname, CURRENT_TIMESTAMP);
   SET temp_id = IDENTITY();
   INSERT INTO ADDRESSES VALUES (DEFAULT, temp_id, address);
 END
```

The BNF for variable declaration is given below:

```
<SQL variable declaration> ::= DECLARE <variable name list> <data type> [DEFAULT
<default value>]
```

```
<variable name list> ::= <variable name> [ { <comma> <variable name> }... ]
```

Examples of variable declaration are given below. Note that in a DECLARE statement with multiple comma-separated variable names, the type and the default value applies to all the variables in the list:

```
 BEGIN ATOMIC
   DECLARE temp_zero DATE;
   DECLARE temp_one, temp_two INTEGER DEFAULT 2;
   DECLARE temp_three VARCHAR(20) DEFAULT 'no name';
   -- more statements ...
   SET temp_zero = DATE '2010-03-18';
   SET temp_two = 5;
   -- more statements ...
 END
```

Handlers

A `<handler declaration>` defines the course of action when an exception or warning is raised during the execution of the compound statement. A compound statement may have one or more handler declarations. These handlers become active when code execution enters the compound statement block and remain active in any sub-block and statement within the block. The handlers become inactive when code execution leaves the block.

In the previous example, if an exception is thrown during the execution of either SQL statement, the execution of the compound statement is terminated and the exception is propagated and thrown by the CALL statement for the procedure. A handler declaration can resolve the thrown exception within the compound statement without propagating it, and allow the execution of the <compound statement> to continue.

In the example below, the UNDO handler declaration catches any exception that is thrown during the execution of the compound statement inside the BEGIN / END block. As it is an UNDO handler, all the changes to data performed within the compound statement (BEGIN / END) block are rolled back. The procedure then returns without throwing an exception.

```
CREATE PROCEDURE NEW_CUSTOMER(IN firstname VARCHAR(50), IN lastname VARCHAR(50), IN address
 VARCHAR(100))
   MODIFIES SQL DATA
 label_one: BEGIN ATOMIC
   DECLARE temp_id INTEGER;
   DECLARE UNDO HANDLER FOR SQLEXCEPTION LEAVE label_one;
   INSERT INTO CUSTOMERS VALUES (DEFAULT, firstname, lastname, CURRENT_TIMESTAMP);
   SET temp_id = IDENTITY();
   INSERT INTO ADDRESSES VALUES (DEFAULT, temp_id, address);
 END
```

Other types of hander are CONTINUE and EXIT handlers. A CONTINUE handler ignores any exception and proceeds to the next statement in the block. An EXIT handler terminates execution without undoing the data changes performed by the previous (successful) statements.

The conditions can be general conditions, or specific conditions. Among general conditions that can be specified, SQLEXCEPTION covers all exceptions, SQLWARNING covers all warnings, while NOT FOUND covers the not-

found condition, which is raised when a DELETE, UPDATE, INSERT or MERGE statement completes without actually affecting any row. Alternatively, one or more specific conditions can be specified (separated with commas) which apply to specific exceptions or warnings or classes or exceptions or warnings. A specific condition is specified with `SQLSTATE <value>`, for example SQLSTATE 'W_01003' specifies the warning raised after a SQL statement is executed which contains an aggregate function which encounters a null value during execution. An example is given below which activates the handler when either of the two warnings is raised:

```
DECLARE UNDO HANDLER FOR SQLSTATE 'W_01003', 'W_01004' LEAVE label_one;
```

The BNF for `<handler declaration>` is given below:

```
<handler declaration> ::= DECLARE {UNDO | CONTINUE | EXIT} HANDLER FOR {SQLEX-
CEPTION | SQLWARNING | NOT FOUND} | { SQL_STATE <state value> [, ...]} [<SQL
procedure statement>];
```

A handler declaration may specify an SQL procedure statement to be performed when the handler is activated. When an exception occurs, the example below performs the UNDO as in the previous example, then inserts the (invalid) data into a separate table.

```
DECLARE UNDO HANDLER FOR SQLEXCEPTION
    INSERT INTO invalid_customers VALUES(firstanme, lastname, address);
```

The `<SQL procedure statement>` is required by the SQL Standard but is optional in HyperSQL. If the execution of the `<SQL procedure statement>` specified in the handler declaration throws an exception itself, then it is handled by the handlers that are currently active. The `<SQL procedure statement>` can itself be a compound statement with its own handlers.

Assignment Statement

The SET statement is used for assignment. It can be used flexibly with rows or single values. The BNF is given below:

```
<assignment statement> ::= <singleton variable assignment> | <multiple variable
assignment>
```

```
<singleton variable assignment> ::= SET <assignment target> <equals operator>
<assignment source>
```

```
<multiple variable assignment> ::= SET (<variable or parameter>, ...) = <row
value expression>
```

In the example below, the result of the SELECT is assigned to two OUT or INOUT arguments. The SELECT must return one row. If it returns more than one, an exception is raised. If it returns no row, no change is made to ARG1 and ARG2.

```
SET (arg1, arg2) = (SELECT col1, col2 FROM atable WHERE id = 10);
```

In the example below, the result of a function call is assigned to VAR1.

```
SET var1 = SQRT(var2);
```

Select Statement : Single Row

A special form of SELECT can also be used for assigning values from a query to one or more arguments or variables. This works similar to a SET statement that has a SELECT statement as the source.

SELECT : SINGLE ROW

select statement: single row

```
<select statement: single row> ::= SELECT [ <set quantifier> ] <select list>
INTO <select target list> <table expression>

<select target list> ::= <target specification> [ { <comma> <target specifica-
tion> }... ]
```

Retrieve values from a specified row of a table and assign the fields to the specified targets. The example below has an identical effect to the example of SET statement given above.

```
SELECT col1, col2 INTO arg1, arg2 FROM atable WHERE id = 10;
```

Formal Parameters

Each parameter of a procedure can be defined as IN, OUT or INOUT. An IN parameter is an input to the procedure and is passed by value. The value cannot be modified inside the procedure body. An OUT parameter is a reference for output. An INOUT parameter is a reference for both input and output. An OUT or INOUT parameter argument is passed by reference, therefore only a dynamic parameter argument or a variable within an enclosing procedure can be passed for it. The assignment statement is used to assign a value to an OUT or INOUT parameter.

In the example below, the procedure is declared with an OUT parameter.

```
CREATE PROCEDURE new_customer(OUT newid INT, IN firstname VARCHAR(50), IN lastname VARCHAR(50),
 IN address VARCHAR(100))
  MODIFIES SQL DATA
  BEGIN ATOMIC
    DECLARE temp_id INTEGER;
    INSERT INTO CUSTOMERS VALUES (DEFAULT, firstname, lastname, CURRENT_TIMESTAMP);
    SET temp_id = IDENTITY();
    INSERT INTO ADDRESSES VALUES (DEFAULT, temp_id, address);
    SET newid = temp_id;
  END
```

In the SQL session, or in the body of another stored procedure, a variable must be assigned to the OUT parameter. After the procedure call, this variable will hold the new identity value that was generated inside the procedure.

In the example below, a session variable, the_new_id is declared. After the call to new_customer, the value for the identity is stored in the_new_id variable. This is returned via the next CALL statement. Alternatively, the_new_id can be used as an argument to another CALL statement.

```
DECLARE the_new_id INT DEFAULT NULL;
CALL new_customer(the_new_id, 'John', 'Smith', '10 Parliament Square');
CALL the_new_id;
```

Iterated Statements

Various iterated statements can be used in routines. In these statements, the <SQL statement list> consists of one or more SQL statements. The <search condition> can be any valid SQL expression of BOOLEAN type.

```
<loop statement> ::= [ <beginning label> <colon> ] LOOP <SQL statement list>
END LOOP [ <ending label> ]

<while statement> ::= [ <beginning label> <colon> ] WHILE <search condition> DO
<SQL statement list> END WHILE [ <ending label> ]

<repeat statement> ::= [ <beginning label> <colon> ]

REPEAT <SQL statement list> UNTIL <search condition> END REPEAT [ <ending label>
```

In the example below, a multiple rows are inserted into a table in a WHILE loop:

```
loop_label: WHILE my_var > 0 DO
  INSERT INTO CUSTOMERS VALUES (DEFAULT, my_var);
  SET my_var = my_var - 1;
  IF my_var = 10 THEN SET my_var = 8; END IF;
  IF my_var = 22 THEN LEAVE loop_label; END IF;
END WHILE loop_label;
```

Conditional Statements

There are two types of CASE ... WHEN statement and the IF ... THEN statement.

CASE WHEN

case when statement

The simple case statement uses a `<case operand>` as the predicand of one or more predicates. For the right part of each predicate, it specifies one or more SQL statements to execute if the predicate evaluates TRUE. If the ELSE clause is not specified, at least one of the search conditions must be true, otherwise an exception is raised.

```
<simple case statement> ::= CASE <case operand> <simple case statement when
clause>... [ <case statement else clause> ] END CASE

<simple case statement when clause> ::= WHEN <when operand list> THEN <SQL
statement list>

<case statement else clause> ::= ELSE <SQL statement list>
```

A skeletal example is given below. The variable var_one is first tested for equality with 22 or 23 and if the test evaluates to TRUE, then the INSERT statement is performed and the statement ends. If the test does not evaluate to TRUE, the next condition test, which is an IN predicate, is performed with var_one and so on. The statement after the ELSE clause is performed if none the previous tests returns TRUE.

```
CASE var_one
  WHEN 22, 23 THEN INSERT INTO t_one ...;
  WHEN IN (2, 4, 5) THEN DELETE FROM t_one WHERE ...;
  ELSE UPDATE t_one ...;
  END CASE
```

The searched case statement uses one or more search conditions, and for each search condition, it specifies one or more SQL statements to execute if the search condition evaluates TRUE. An exception is raised if there is no ELSE clause and none of the search conditions evaluates TRUE.

```
<searched case statement> ::= CASE <searched case statement when clause>...
[ <case statement else clause> ] END CASE

<searched case statement when clause> ::= WHEN <search condition> THEN <SQL
statement list>
```

The example below is partly a rewrite of the previous example, but a new condition is added:

```
CASE WHEN var_one = 22 OR var_one = 23 THEN INSERT INTO t_one ...;
  WHEN var_one IN (2, 4, 5) THEN DELETE FROM t_one WHERE ...;
  WHEN var_two IS NULL THEN UPDATE t_one ...;
  ELSE UPDATE t_one ...;
  END CASE
```

IF

if statement

The if statement is very similar to the searched case statement. The difference is that no exception is raised if there is no ELSE clause and no search condition evaluates TRUE.

```
<if statement> ::= IF <search condition> <if statement then clause> [ <if
statement elseif clause>... ] [ <if statement else clause> ] END IF

<if statement then clause> ::= THEN <SQL statement list>

<if statement elseif clause> ::= ELSEIF <search condition> THEN <SQL statement
list>

<if statement else clause> ::= ELSE <SQL statement list>
```

Return Statement

The RETURN statement is required and used only in functions. The body of a function is either a RETURN statement, or a compound statement that contains a RETURN statement.

RETURN

return statement

```
<return statement> ::= RETURN <return value>

<return value> ::= <value expression> | NULL
```

Return a value from an SQL function. If the function is defined as RETURNS TABLE, then the value is a TABLE expression such as RETURN TABLE(SELECT ...) otherwise, the value expression can be any scalar expression. In the examples below, the same function is written with or without a BEGIN END block. In both versions, the RETURN value is a scalar expression.

```
CREATE FUNCTION an_hour_before_max (e_type INT)
  RETURNS TIMESTAMP
  RETURN (SELECT MAX(event_time) FROM atable WHERE event_type = e_type) - 1 HOUR

CREATE FUNCTION an_hour_before_max (e_type INT)
  RETURNS TIMESTAMP
  BEGIN ATOMIC
    DECLAR max_event TIMESTAMP;
    SET max_event = SELECT MAX(event_time) FROM atable WHERE event_type = e_type;
    RETURN max_event - 1 HOUR;
  END
```

Control Statements

In addition to the RETURN statement, the following statements can be used in specific contexts.

ITERATE STATEMENT

The ITERATE statement can be used to cause the next iteration of a labeled iterated statement (a WHILE, REPEAT or LOOP statement). It is similar to the "continue" statement in C and Java.

```
<iterate statement> ::= ITERATE <statement label>
```

LEAVE STATEMENT

The LEAVE statement can be used to leave a labeled block. When used in an iterated statement, it is similar to the "break" statement is C and Java. But it can be used in compound statements as well.

```
<leave statement> ::= LEAVE <statement label>
```

Signal and Resignal Statements

The SIGNAL statement is used to throw an exception (or force an exception). When invoked, any exception handler for the given exception is in turn invoked. If there is no handler, the exception is propagated to the enclosing context.

```
<signal statement> ::= SIGNAL SQL_STATE <state value>
```

The RESIGNAL statement is used to throw an exception from an exception handler's `<SQL procedure state-ment>`, in effect propagating the exception to the enclosing context without further action by the currently active handlers.

```
<resignal statement> ::= RESIGNAL SQL_STATE <state value>
```

Routine Polymorphism

More than one version of a routine can be created.

For procedures, the different versions must have different parameter counts. When the procedure is called, the parameter count determines which version is called.

For functions, the different versions can have the same or different parameter counts. When the parameter count of two versions of a function is the same, the type of parameters must be different. The best matching version of the function is called, according to both the parameter count and parameter types.

Two versions of an overloaded function are given below. One version accepts TIMESTAMP while the other accepts TIME arguments.

```
CREATE FUNCTION an_hour_before_or_now(t TIMESTAMP)
  RETURNS TIMESTAMP
  IF t > CURRENT_TIMESTAMP THEN
    RETURN CURRENT_TIMESTAMP;
  ELSE
    RETURN t - 1 HOUR;
  END IF

CREATE FUNCTION an_hour_before_or_now(t TIME)
  RETURNS TIME
  CASE t
    WHEN > CURRENT_TIME THEN
      RETURN CURRENT_TIME;
    WHEN >= TIME'01:00:00' THEN
      RETURN t - 1 HOUR;
    ELSE
      RETURN CURRENT_TIME;
  END CASE
```

more ..

Returning Data From Routines

The OUT or INOUT parameters of a procedure are used to assign simple values to dynamic parameters or to variables in the calling context. In addition, a SQL/PSM procedure may return result sets to the calling context. These result

sets are dynamic in the sense that a procedure may return a different number of result sets or none at all in different invocations.

The SQL Standard uses a mechanism called CURSORS for accessing and modifying rows of a result set one by one. This mechanism is absolutely necessary when the database is accessed from an external application program. The JDBC ResultSet interface allows this method of access from Java programs and is supported by HyperSQL.

The SQL Standard uses cursors within the body of a procedure to return result sets. It specifies a somewhat complex mechanism to allow access to these cursors from the calling contexts. HyperSQL does not support access to such result sets within a calling SQL/PSM procedure. This is considered redundant as all operations on data can be performed with non-cursor SQL statements.

(feature to be implemented) HyperSQL will support returning single or multiple result sets from SQL/PSM procedures only via the JDBC CallableStatement interface. Cursors are declared and opened within the body of the procedure. No further operation is performed on the cursors within the procedure. When the execution of the procedure is complete, the cursors become available as Java ResultSet objects via the CallableStatement instance that called the SQL/PSM procedure.

Currently, a single result can be returned from FUNCTION routines, when the function is defined as RETURNS TABLE (..)

To return a table from a SELECT statement, you should use a return statement such as RETURN TABLE(SELECT ...); in a SQL/PSM function. A Java function should return a JDBCResultSet instance. For an example of how to construct a JDBCResultSet for this purpose, see the source code for the org.hsqldb.jdbc.JDBCArray class.

The JDBC CallableStatement class is used with the SQL statement `CALL <routine name> ( <argument 1>, ... )` to call both functions and procedures. The `getXXX()` methods can be used to retrieve INOUT or OUT arguments after the call. The `getResultSet()` call can be used to access the ResultSet returned from a function that returns a result set.

Java Language Routines (SQL/JRT)

The body of a Java language routine is a static method of a Java class, specified with a fully qualified method name in the routine definition.

In the example below, the static method named `toZeroPaddedString` is specified to be called when the function is invoked.

```
CREATE FUNCTION zero_pad(x BIGINT, digits INT, maxsize INT)
  RETURNS CHAR VARYING(100)
  NO SQL
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME
  'CLASSPATH:org.hsqldb.lib.StringUtil.toZeroPaddedString'
```

The signature of the Java method (used in the Java code but not in SQL code to create the function) is given below:

```
public static String toZeroPaddedString(long value, int precision, int maxSize)
```

The parameter and return types and of the SQL routine definition must match those of the Java method according to the table below:

SMALLINT	Short or Short
INT	int or Integer

BIGINT	long or Long
NUMERIC or DECIMAL	BigDecimal
FLOAT or DOUBLE	Double or Double
CHAR or VARCHAR	String
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BINARY	Byte[]
BOOLEAN	boolean or Boolean
ARRAY of any type	java.sql.Array
TABLE	java.sql.ResultSet

If the specified Java method is not found or its parameters and return types do not match the definition, an exception is raised. If more than one version of the Java method exist, then the one with matching parameter and return types is found and registered. If two "equivalent" methods exist, the first one is registered. (This situation arises only when a parameter is a primitive in one version and an Object in another version, e.g. `long` and `java.lang.Long`.).

When the Java method of an SQL/JRT routine returns a value, it should be within the size and precision limits defined in the return type of the SQL-invoked routine, otherwise an exception is raised. The scale difference are ignored and corrected. For example, in the above example, the `RETURNS CHAR VARYING(100)` clause limits the length of the strings returned from the Java method to 100. But if the number of digits after the decimal point (scale) of a returned BigDecimal value is larger than the scale specified in the RETURNS clause, the decimal fraction is silently truncated and no exception of warning is raised.

Polymorphism

If two versions of the same SQL invoked routine with different parameter types are required, they can be defined to point to the same method name or different method names, or even methods in different classes. In the example below, the first two definitions refer to the same method name in the same class. In the Java class, the two static methods are defined with corresponding method signatures.

In the third example, the Java function returns a result set and the SQL declaration includes RETURNS TABLE.

```
CREATE FUNCTION an_hour_before_or_now(t TIME)
  RETURNS TIME
  NO SQL
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'CLASSPATH:org.npo.lib.nowLessAnHour'

CREATE FUNCTION an_hour_before_or_now(t TIMESTAMP)
  RETURNS TIMESTAMP
  NO SQL
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'CLASSPATH:org.npo.lib.nowLessAnHour'

CREATE FUNCTION testquery(INTEGER)
  RETURNS TABLE(n VARCHAR(20), i INT)
  READS SQL DATA
  LANGUAGE JAVA
  EXTERNAL NAME 'CLASSPATH:org.hsqldb.test.TestJavaFunctions.getQueryResult'
```

In the Java class:

```
    public static java.sql.Time nowLessAnHour(java.sql.Time value) {
        ...
    }
    public static java.sql.Timestamp nowLessAnHour(java.sql.Timestamp value)
        ...
    }

    public static ResultSet getQueryResult(Connection connection, int i) throws SQLException {
        Statement st = connection.createStatement();
        return st.executeQuery("SELECT * FROM T WHERE I < " + i);
    }
```

Java Language Procedures

Java procedures are defined similarly to functions. The differences are:

- The return type of the Java static method must be void.

- If a parameter is defined as OUT or INOUT, the corresponding Java static method parameter must be defined as an array of the JDBC non-primitive type.

- When the Java static method is invoked, the OUT and INOUT arguments are passed as a single-element array.

- The static method can modify the OUT or INOUT param by assigning a value to the sole element of the argument array.

- If the procedure contains SQL statements, only statements for data access and manipulation are allowed. The java method should not perform commit or rollback. The SQL statements should not change the session settings and should not include statements at create or modify tables definitions or other database objects. These rules are generally enforced by the engine, but additional enforcement may be added in future versions

An example of a procedure definition is given below:

```
CREATE PROCEDURE new_customer(firstname VARCHAR(50), lastname  VARCHAR(50), address VARCHAR(100))
  MODIFIES SQL DATA
  LANGUAGE JAVA
  EXTERNAL NAME 'CLASSPATH:org.hsqldb.test.Test01.newCustomerProcedure'
```

Legacy Support

The legacy HyperSQL statement, CREATE ALIAS <name> FOR <fully qualified Java method name> is no longer supported directly. It is supported when importing databases and translates to a special CREATE FUNCTION <name> statement that creates the function in the PUBLIC schema.

The direct use of a Java method as a function is still supported but deprecated. It is internally translated to a special CREATE FUNCTION statement where the name of the function is the double quoted, fully qualified name of the Java method used.

SQL Language Aggregate Functions

HyperSQL adds an extension to the SQL Standard to allow user defined aggregate functions. A user defined aggregate function has a single parameter when it is used in SQL statements. Unlike the predefined aggregate functions, the keyword DISTINCT cannot be used when a user defined aggregate function is invoked. Like all user defined functions, an aggregate function belongs to a schema and can be polymorphic.

A user defined aggregate function can be used in SQL statements where a predefined aggregate function is allowed.

Definition of Aggregate Functions

An aggregate function is always defined with 4 parameters. The first parameter is the parameter that is used when the function is invoked in SQL statements, the rest of the parameter are invisible to the invoking statement. The type of the first parameter is user defined. The type of the second parameter must be BOOLEAN. The third and fourth parameters have user defined types and must be defined as INOUT parameters. The defined return type of the function determines the type of the value returned when the function is invoked.

CREATE AGGREGATE FUNCTION

user defined aggregate function definition

Aggregate function definition is similar to normal function definition and has the mandatory `<returns clause>`. The BNF is given below.

```
<user defined aggregate function> ::= CREATE AGGREGATE FUNCTION <schema quali-
fied routine name> <SQL aggregate parameter declaration list> <returns clause>
<routine characteristics> <routine body>
```

The parameter declaration list BNF is given below. The type of the first parameter is used when the function is invoked as part of an SQL statement. When multiple versions of a function are required, each version will have the first parameter of a different type.

```
<SQL aggregate declaration list> ::= <left paren> [IN] [ <SQL parameter name> ]
<parameter type> <comma> [IN] [ <SQL parameter name> ] BOOLEAN <comma> INOUT
[ <SQL parameter name> ] <parameter type> <comma> INOUT [ <SQL parameter name> ]
<parameter type> <right paren>
```

The return type is user defined. This is the type of the resulting value when the function is called. Usually an aggregate function is defined with CONTAINS SQL, as it normally does not read the data in database tables, but it is possible to define the function with READS SQL DATA and access the database tables.

HyperSQL invokes the aggregate function, with all the arguments set, once per each row in order to compute the values. Finally, it invokes the function once more to return the final result.

In the computation phase, the first argument is the value of the user argument as specified in the SQL statement, computed for the current row. The second argument is the boolean FALSE. The third and fourth argument values are initially null, but they can be updated in the body of the function during each invocation. The third and fourth arguments act as registers and hold their values between invocations. The return value of the function is ignored during the computation phase (when the second parameter is FALSE).

After the computation phase, the function is invoked once more to get the final result. In this invocation, the first argument is NULL and the second argument is boolean TRUE. The third and fourth arguments hold the values they held at the end of the last invocation. The value returned by the function in this invocation is used as the result of the aggregate function computation in the invoking SQL statement. In SQL queries with GROUP BY, the call sequence is repeated for each separate group.

SQL PSM Aggregate Functions

The example below features a user defined version of the Standard `AVG(<value expression>)` aggregate function for INTEGER input and output types. This function behaves differently from the Standard AVG function as it returns 0 when all the input values are null.

```
CREATE AGGREGATE FUNCTION udavg(IN x INTEGER, IN flag BOOLEAN, INOUT addup BIGINT, INOUT counter
 INT)
  RETURNS  INTEGER
```

```
  CONTAINS SQL
  BEGIN ATOMIC
    IF flag THEN
      RETURN addup / counter;
    ELSE
      SET counter = COALESCE(counter, 0) + 1;
      SET addup = COALESCE(addup, 0) || COALESCE(x, 0);
      RETURN NULL;
    END IF;
  END
```

The user defined aggregate function is used in a select statement in the example below. Only the first parameter is visible and utilised in the select statement.

```
SELECT udavg(id) FROM customers GROUP BY lastname;
```

In the example below, the function returns an array that contains all the values passed for the aggregated column. For use with longer arrays, you can optimise the function by defining a larger array in the first iteration, and using the TRIM_ARRAY function on the RETURN to cut the array to size :

```
CREATE AGGREGATE FUNCTION array_aggregate(IN val VARCHAR(100), IN flag boolean, INOUT buffer
 VARCHAR(100) ARRAY, INOUT counter INT)
  RETURNS VARCHAR(100) ARRAY
  CONTAINS SQL
  BEGIN ATOMIC
    IF flag THEN
      RETURN buffer;
    ELSE
      IF val IS NULL THEN RETURN NULL; END IF;
      IF counter IS NULL THEN SET counter = 0; END IF;
      SET counter = counter + 1;
      IF counter = 1 THEN SET buffer = ARRAY[val];
      ELSE SET buffer[counter] = val; END IF;
      RETURN NULL;
    END IF;
  END
```

The tables and data for the select statement below are created with the DatabaseManager or DatabaseManagerSwing GUI apps. Part of the output is shown. Each row of the output includes an array containing the values for the invoices for each customer.

```
SELECT ID, FIRSTNAME, LASTNAME, ARRAY_AGGREGATE(CAST(INVOICE.TOTAL AS VARCHAR(100)))
  FROM customer JOIN INVOICE ON ID =CUSTOMERID
  GROUP BY ID, FIRSTNAME, LASTNAME

11 Susanne    Karsen     ARRAY['3988.20']
12 John       Peterson   ARRAY['2903.10','4382.10','4139.70','3316.50']
13 Michael    Clancy     ARRAY['6525.30']
14 James      King       ARRAY['3665.40','905.10','498.00']
18 Sylvia     Clancy     ARRAY['634.20','4883.10']
20 Bob        Clancy     ARRAY['3414.60','744.60']
```

Java Aggregate Functions

A Java aggregate function is defined similarly to PSM functions, apart from the routine body, which is defined as EXTERNAL NAME ... The Java function signature must follow the rules for both nullable and INOUT parameters, therefore:

No agrument is defined as a primitive or primitive array type. This allows nulls to be passed to the function. The second and third arguments must be defined as arrays of the JDBC non-primitive types listed in the table in the previous section.

In the example below, a user-defined aggregate function for geometric mean is defined.

```
CREATE AGGREGATE FUNCTION geometric_mean(IN val DOUBLE, IN flag BOOLEAN, INOUT register DOUBLE,
 INOUT counter INT)
 RETURNS DOUBLE
 NO SQL
 LANGUAGE JAVA
 EXTERNAL NAME 'CLASSPATH:org.hsqldb.test.Test01.geometricMean'
```

The Java function definition is given below:

```
public static Double geometricMean(Double in, Boolean flag,
        Double[] register, Integer[] counter) {

    if (flag) {
        if (register[0] == null) { return null; }
        double a = register[0].doubleValue();
        double b = 1 / (double) counter[0];
        return Double.valueOf(java.lang.Math.pow(a, b));
    }
    if (in == null) { return null; }
    if (in.doubleValue() == 0) { return null; }
    if (register[0] == null) {
        register[0] = in;
        counter[0]  = Integer.valueOf(1);
    } else {
        register[0] = Double.valueOf(register[0].doubleValue() * in.doubleValue());
        counter[0] = Integer.valueOf(counter[0].intValue() + 1);
    }
    return null;
}
```

In a select statement, the function is used like built in aggregate functions:

```
SELECT geometric_mean(age) FROM  FROM customer
```

Routine Definition

As discussed in the previous pages, routine definition has several mandatory or optional clauses. The complete BNF supported by HyperSQL and the remaining clauses are documented in this section.

CREATE FUNCTION

CREATE PROCEDURE

routine definition

Routine definition is similar for procedures and functions. A function definition has the mandatory `<returns clause>` which is discussed later. The description given so far covers the essential elements of the specification with the BNF given below.

`<schema procedure> ::= CREATE PROCEDURE <schema qualified routine name> <SQL parameter declaration list> <routine characteristics> <routine body>`

`<schema function> ::= CREATE FUNCTION <schema qualified routine name> <SQL parameter declaration list> <returns clause> <routine characteristics> <routine body>`

Parameter declaration list has been described above. For SQL/JRT routines, the `<SQL parameter name>` is optional while for SQL/PSM routines, it is required. If the `<parameter mode>` of a parameter is OUT or INOUT, it must be specified. The BNF is given below:

```
<SQL parameter declaration list> ::= <left paren> [ <SQL parameter declaration>
[ { <comma> <SQL parameter declaration> }... ] ] <right paren>

<SQL parameter declaration> ::= [ <parameter mode> ] [ <SQL parameter name> ]
<parameter type>

<parameter mode> ::= IN | OUT | INOUT

<parameter type> ::= <data type>
```

Return Value and Table Functions

RETURNS

returns clause

The `<returns clause>` specifies the type of the return value of a function. For all SQL/PSM functions and ordinary SQL/JRT functions, this is simply a type definition which can be a built-in type, a DOMAIN type or a DISTINCT type, or alternatively, a TABLE definition. For example, RETURNS INTEGER.

For a SQL/JRT function, it is possible to define a `<returns table type>` for a Java method that returns a `java.sql.ResultSet` object. Such SQL/JRT functions are called *table functions*. Table functions are used differently from normal functions. A table function can be used in an SQL query expression exactly where a normal table or view is allowed. At the time of invocation, the Java method is called and the returned ResultSet is transformed into an SQL table. The column types of the declared TABLE must match those of the ResultSet, otherwise an exception is raised at the time of invocation.

If a `<returns table type>` is defined for an SQL/PSM function, the following expression is used inside the function to return a table: RETURN TABLE (`<query expression>`); In the example blow, a table with two columns is returned.

```
RETURN TABLE ( SELECT a, b FROM atable WHERE e = 10 );
```

If a JDBC `CallableStatement` is used to CALL the function, the table returned from the function call is returned and can be accessed with the `getResultSet()` method of the `CallableStatement`.

```
<returns clause> ::= RETURNS <returns type>

<returns type> ::= <returns data type> | <returns table type>

<returns table type> ::= TABLE <table function column list>

<table function column list> ::= <left paren> <table function column list
element> [ { <comma> <table function column list element> } ... ] <right paren>

<table function column list element> ::= <column name> <data type>

<returns data type> ::= <data type>
```

routine body

routine body

Routine body is either one or more SQL statements or a Java reference, as described. The user that defines the routine by issuing the CREATE FUNCTION or CREATE SCHEMA command must have the relevant access rights to all tables, sequences, routines, etc. that are accessed by the routine. If another user is given EXECUTE privilege on the routine, then there are two possibilities, depending on the `<rights clause>`. This clause refers to the access rights

that are checked when a routine is invoked. The default is `SQL SECURITY DEFINER`, which means access rights of the definer are used; therefore no extra checks are performed when the other user invokes the routine. The alternative `SQL SECURITY INVOKER` means access rights on all the database objects referenced by the routine are checked for the invoker. This alternative is not supported by HyperSQL.

```
<routine body> ::= <SQL routine spec> | <external body reference>

<SQL routine spec> ::= [ <rights clause> ] <SQL routine body>

<rights clause> ::= SQL SECURITY INVOKER | SQL SECURITY DEFINER
```

SQL routine body

SQL routine body

The routine body of a an SQL routine consists of an statement.

```
<SQL routine body> ::= <SQL procedure statement>
```

EXTERNAL NAME

external body reference

External name specifies the qualified name of the Java method associated with this routine. Early releases of HyperSQL 2.0 only supports Java methods within the classpath. The `<external Java reference string>` is a quoted string which starts with CLASSPATH: and is followed by the Java package, class and method names separated with dots. HyperSQL does not currently support the optional `<Java parameter declaration list>`.

```
<external body reference> ::= EXTERNAL NAME <external Java reference string>

<external Java reference string> ::= <jar and class name> <period> <Java method name> [ <Java parameter declaration list> ]
```

Routine Characteristics

The `<routine characteristics>` clause covers several sub-clauses

```
<routine characteristics> ::= [ <routine characteristic>... ]

<routine characteristic> ::= <language clause> | <parameter style clause> |
SPECIFIC <specific name> | <deterministic characteristic> | <SQL-data access
indication> | <null-call clause> | <returned result sets characteristic> |
<savepoint level indication>
```

LANGUAGE

language clause

The `<language clause>` refers to the language in which the routine body is written. It is either SQL or Java. The default is SQL, so JAVA must be specified for SQL/JRT routines.

```
<language clause> ::= LANGUAGE <language name>

<language name> ::= SQL | JAVA
```

The parameter style is not allowed for SQL routines. It is optional for Java routines and, in HyperSQL, the only value allowed is JAVA.

```
<parameter style> ::= JAVA
```

SPECIFIC NAME

specific name

The SPECIFIC <specific name> clause is optional but the engine will creates an automatic name if it is not present. When there are several versions of the same routine, the <specific name> is used in schema manipulation statements to drop or alter a specific version. The <specific name> is a user-defined name. It applies to both functions and procedures. In the examples below, a specific name is specified for each function.

```
CREATE FUNCTION an_hour_before_or_now(t TIMESTAMP)
  RETURNS TIMESTAMP
  NO SQL
  LANGUAGE JAVA PARAMETER STYLE JAVA
  SPECIFIC an_hour_before_or_now_with_timestamp
  EXTERNAL NAME 'CLASSPATH:org.npo.lib.nowLessAnHour'

CREATE FUNCTION an_hour_before_max (e_type INT)
  RETURNS TIMESTAMP SPECIFIC an_hour_before_max_with_int
  RETURN (SELECT MAX(event_time) FROM atable WHERE event_type = e_type) - 1 HOUR
```

DETERMINISTIC

deterministic characteristic

The <deterministic characteristic> clause indicates that a routine is deterministic or not. Deterministic means the routine does not reference random values, external variables, or time of invocation. The default is NOT DETERMINISTIC. It is essential to declare this characteristics correctly for an SQL/JRT routine, as the engine does not know the contents of the Java code, which could include calls to methods returning random or time sensitive values.

```
<deterministic characteristic> ::= DETERMINISTIC | NOT DETERMINISTIC
```

SQL DATA access

SQL DATA access characteristic

The <SQL-data access indication> clause indicates the extent to which a routine interacts with the database or the data stored in the database tables (SQL data). NO SQL means no SQL command is issued in the routine body and can be used only for SQL/JRT functions. CONTAINS SQL means some SQL commands are used, but they do not read or modify the SQL data. READS SQL DATA and MODIFIES SQL DATA are self explanatory.

```
<SQL-data access indication> ::= NO SQL | CONTAINS SQL | READS SQL DATA |
MODIFIES SQL DATA
```

NULL INPUT

null call clause

Null Arguments

The <null-call clause> is used only for functions. If a function returns NULL when any of the calling arguments is null, then by specifying RETURNS NULL ON NULL INPUT, calls to the function are known to be redundant and do not take place when an argument is null. This simplifies the coding of the SQL/JRT Java methods and improves performance at the same time.

```
<null-call clause> ::= RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT
```

SAVEPOINT LEVEL

transaction impact

The `<savepoint level indication>` is used only for procedures and refers to the visibility of existing savepoints within the body of the procedure. If `NEW SAVEPOINT LEVEL` is specified, savepoints that have been declared prior to calling the procedure become invisible within the body of the procedure. HyperSQL's implementation accepts only `NEW SAVEPOINT LEVEL`, which must be specified.

```
<savepoint level indication> ::= NEW SAVEPOINT LEVEL | OLD SAVEPOINT LEVEL
```

DYNAMIC RESULT SETS

returned result sets characteristic

The `<returned result sets characteristic>` is used only for SQL/PSM procedures. The maximum number of result sets that a procedure may return can be specified with the clause below. The default is zero. Details are discussed in the previous sections.

```
<returned result sets characteristic> ::= DYNAMIC RESULT SETS <maximum returned
result sets>
```

Chapter 9. Triggers

Fred Toussi, The HSQL Development Group

$Revision: 3042 $

Overview

Trigger functionality first appeared in SQL:1999. Triggers embody the *live database* concept, where changes in SQL data can be monitored and acted upon. This means each time a DELETE, UPDATE or INSERT is performed, additional actions are taken by the declared triggers. SQL Standard triggers are *imperative* while the *relational* aspects of SQL are *declarative*. Triggers allow performing an arbitrary transformation of data that is being updated or inserted, or to prevent insert, updated or deletes, or to perform additional operations.

Some bad examples of SQL triggers in effect enforce an "integrity constraint" which would better be expressed as a CHECK constraint. A trigger that causes an exception if the value inserted in a column is negative is such an example. A check constraint that declares CHECK VALUE >= 0 (declarative) is a better way of expressing an integrity constraint than a trigger that throws an exception if the same condition is false.

Usage constraints cannot always be expressed by SQL's integrity constraint statements. Triggers can enforce these constraints. For example, it is may be possible to write a check constraint that prevents data from being added, or modified on weekends. But it is not possible to use a check constraint to prevent deletes. A trigger can be used to enforce the time when each operation is allowed.

A trigger can modify the values that are inserted into the database, instead of rejecting them. For example, a badly formatted string can be cleaned up by a trigger before INSERT.

Triggers can also perform additional data changes, for example inserting an additional row into a different table for data audits.

A trigger is declared to activate when an UPDATE, INSERT or DELETE action is performed on a table. These actions may be direct or indirect. Indirect actions may arise from CASCADE actions of FOREIGN KEY constraints, or from data change statements performed on a VIEW that is based on the table that in.

It is possible to declare multiple triggers on a single table. The triggers activate one by one according to the order in which they were defined.

A row level trigger allows access to the deleted or inserted rows. For UPDATE actions there is both an old and new version of each row. A trigger can be specified to activate before or after the action has been performed. Triggers that are performed after the action cannot modify the rows that have been modified. These triggers can perform other actions, such as inserting rows into other tables. Triggers that are performed before the action can modify the inserted or updated rows but not the deleted rows.

A TRIGGER that is declared on a VIEW, is an INSTEAD OF trigger. This term means when an INSERT, UPDATE or DELETE statement is executed, the trigger action is all that is performed, and no further data change takes place on the VIEW. The trigger action can include all the statements that are necessary to change the data in the tables that underlie the VIEW. With the use of INSTEAD OF triggers a read-only view can effectively become updatable or insertable-into.

Trigger Properties

A trigger is declared on a specific table or view. Various trigger properties determine when the trigger is executed and how.

Trigger Event

The trigger event specifies the type of SQL statement that causes the trigger to execute. Each trigger is specified to execute when an INSERT, DELETE or UPDATE takes place.

The event can be filtered by two separate means. For all triggers, the WHEN clause can specify a condition against the rows that are the subject of the trigger, together with the data in the database. For example, a trigger can activate when the size of a table becomes larger than a certain amount. Or it can activate when the values in the rows being modified satisfy certain conditions.

An UPDATE trigger can be declared to execute only when certain columns are the subject of an update statement. For example, a trigger declared as AFTER UPDATE OF (datecolumn) will activate only when the UPDATE statement that is executed includes the column, datecolumn, as one of the columns specified in its SET statements.

Granularity

A statement level trigger is performed once for the executed SQL statement and is declared as FOR EACH STATEMENT.

A row level trigger is performed once for each row that is modified during the execution of an SQL statement and is declared as FOR EACH ROW. Note that an SQL statement can INSERT, UPDATE or DELETE zero or more rows.

If a statement does not apply to any row, then the trigger is not executed.

If FOR EACH ROW or FOR EACH STATEMENT is not specified, then the default is FOR EACH STATEMENT.

The granularity dictates whether the REFERENCING clause can specify OLD ROW, NEW ROW, or OLD TABLE, NEW TABLE.

A trigger declared as FOR EACH STATEMENT can only be an AFTER trigger.

Trigger Action Time

A trigger is executed BEFORE, AFTER or INSTEAD OF the trigger event.

INSTEAD OF triggers are allowed only when the trigger is declared on a VIEW. With this type of trigger, the event (SQL statement) itself is not executed, only the trigger.

BEFORE or AFTER triggers are executed just before or just after the execution of the event. For example, just before a row is inserted into a table, the BEFORE trigger is activated, and just after the row is inserted, the AFTER trigger is executed.

BEFORE triggers can modify the row that is being inserted or updated. AFTER triggers cannot modify rows. They are usually used to perform additional operations, such as inserting rows into other tables.

A trigger declared as FOR EACH STATEMENT can only be an AFTER trigger.

References to Rows

If the old rows or new rows are referenced in the SQL statements in the trigger action, they must have names. The REFERENCING clause is used to give names to the old and new rows. The clause, REFERENCING OLD | NEW

TABLE is used for statement level triggers. The clause, REFERENCING OLD | NEW ROW is used for row level triggers. If the old rows or new rows are referenced in the SQL statements in the trigger action, they must have names. In the SQL statements, the columns of the old or new rows are qualified with the specified names.

Trigger Condition

The WHEN clause can specify a condition for the columns of the row that is being changed. Using this clause you can simply avoid unnecessary trigger activation for rows that do not need it.

For UPDATE trigger, you can specify a list of columns of the table. If a list of columns is specified, then if the UPDATE statement does not change the columns with SET clauses, then the trigger is not activated at all.

Trigger Action in SQL

The trigger action specifies what the trigger does when it is activated. This is usually written as one or more SQL statements.

When a row level trigger is activated, there is an OLD ROW, or a NEW ROW, or both. An INSERT statement supplies a NEW ROW row to be inserted into a table. A DELETE statement supplied an OLD ROW be deleted. An UPDATE statement supplies both OLD ROW and NEW ROW that represent the updated rows before and after the update. The REFERENCING clause gives names to these rows, so that the rows can be referenced in the trigger action.

In the example below, a name is given to the NEW ROW and it is used both in the WHEN clause and in the trigger action SQL to insert a row into a triglog table after each row insert into the testtrig table.

```
create trigger trig after insert on testtrig
   referencing new row as newrow
   for each row when (newrow.id > 1)
   insert into triglog values (newrow.id, newrow.data, 'inserted')
```

In the example blow, the trigger code modifies the updated data if a condition is true. This type of trigger is useful when the application does not perform the necessary checks and modifications to data.

```
create trigger t before update on customer
   referencing new as newrow for each row
   begin atomic
   if length(newrow.firstname ) > 10 then
   set newrow.firstname = lower(newrow.firstname);
   end if;
   end
```

Trigger Action in Java

A trigger action can be written as a Java class that implements the org.hsqldb.Trigger interface. This interface has a single method which is called when the trigger is activated, either before or after the event. When the method is called by the engine, it supplies the name of the trigger (as name argument), the name of the table (as table argument), the OLD ROW (as row1 argument) and the NEW ROW (as row2 argument). The row1 argument is null for row level INSERT triggers. The row2 argument is null for row level DELETE triggers. For table level triggers, both arguments are null (that is, there is no access to the data). The triggerType argument is one of the constants in the org.hsqldb.Trigger interface which indicate the type of trigger, for example, INSERT_BEFORE_ROW or UPDATE_AFTER_ROW.

The Java class for the trigger can be reused for several triggers on different tables. The method code can distinguish between the different tables and triggers using the supplied arguments and take appropriate action.

```
    fire (int triggerType, String name, String table, Object row1[], Object row2[])
```

The Java method for a synchronous trigger (see below) can modify the values in row2 in a BEFORE trigger. Such modifications are reflected in the row that is being inserted or updated. Any other modifications are ignored by the engine.

A Java trigger that uses an instance of `org.hsqldb.Trigger` has two forms, synchronous, or asynchronous (immediate or queued). By default, or when QUEUE 0 is specified, the action is performed immediately by calling the Java method. This is similar to SQL trigger actions. When QUEUE n is specified with n larger than 0, the engine uses a separate thread to execute the Java method, using a queue with the size n. For certain applications, such as real-time systems this allows asynchronous notifications to be sent by the trigger event, without introducing delays in the engine. With asynchronous triggers, an extra parameter, NOWAIT can be used in trigger definition. This overcomes the queue full condition. In this mode, old calls that are still in the queue are discarded one by one and replaced with new calls.

Java triggers can modify the row data. They should not be used to modify the database, e.g. insert new rows, etc.

For sample trigger classes and test code see, org.hsqldb.sample.TriggerSample, org.hsqldb.test.TestTriggers, org.hsqldb.test.TriggerClass and the associated text script TestTriggers.txt in /testrun/hsqldb/ directory. In the example below, the trigger is activated only if the update statement includes SET clauses that modify any of the specified columns (c1, c2, c3). Furthermore, the trigger is not activated if the c2 column in the updated row is null.

```
create trigger trigbur before update of c1, c2, c3 on testtrig
   referencing new row as newrow
   for each row when (newrow.c2 is not null)
   call "org.hsqldb.test.TriggerClass"
```

Java functions can be called from an SQL trigger. So it is possible to define the Java function to perform any external communication that are necessary for the trigger, and use SQL code for checks and alterations to data.

```
create trigger t before update on customer
   referencing new as newrow for each row
   begin atomic
   if length(newrow.firstname ) > 10 then
   call my_java_function(newrow.firstname, newrow.lastname);
   end if;
   end
```

Trigger Creation

CREATE TRIGGER

trigger definition

```
<trigger definition> ::= CREATE TRIGGER <trigger name> <trigger action time>
<trigger event> ON <table name> [BEFORE <other trigger name>] [ REFERENCING
<transition table or variable list> ] <triggered action>

<trigger action time> ::= BEFORE | AFTER | INSTEAD OF

<trigger event> ::= INSERT | DELETE | UPDATE [ OF <trigger column list> ]

<trigger column list> ::= <column name list>

<triggered action> ::= [ FOR EACH { ROW | STATEMENT } ] [ <triggered when
clause> ] <triggered SQL statement>

<triggered when clause> ::= WHEN <left paren> <search condition> <right paren>

<triggered SQL statement> ::= <SQL procedure statement> | BEGIN ATOMIC { <SQL
procedure statement> <semicolon> }... END | [QUEUE <integer literal>] [NOWAIT]
CALL <HSQLDB trigger class FQN>
```

```
<transition table or variable list> ::= <transition table or variable>...

<transition table or variable> ::= OLD [ ROW ] [ AS ] <old transition variable
name> | NEW [ ROW ] [ AS ] <new transition variable name> | OLD TABLE [ AS ]
<old transition table name> | NEW TABLE [ AS ] <new transition table name>

<old transition table name> ::= <transition table name>

<new transition table name> ::= <transition table name>

<transition table name> ::= <identifier>

<old transition variable name> ::= <correlation name>

<new transition variable name> ::= <correlation name>
```

Trigger definition is a relatively complex statement. The combination of `<trigger action time>` and `<trigger event>` determines the type of the trigger. Examples include BEFORE DELETE, AFTER UPDATE, INSTEAD OF INSERT. If the optional `[ OF <trigger column list> ]` is specified for an UPDATE trigger, then the trigger is activated only if one of the columns that is in the `<trigger column list>` is specified in the UPDATE statement that activates the trigger.

If a trigger is `FOR EACH ROW`, which is the default option, then the trigger is activated for each row of the table that is affected by the execution of an SQL statement. Otherwise, it is activated once only per statement execution. In the first case, there is a before and after state for each row. For UPDATE triggers, both before and after states exist, representing the row before the update, and after the update. For DELETE, triggers, there is only a before state. For INSERT triggers, there is only an after state. If a trigger is `FOR EACH STATEMENT`, then a transient table is created containing all the rows for the before state and another transient table is created for the after state.

The `[ REFERENCING <transition table or variable> ]` is used to give a name to the before and after data row or table. This name can be referenced in the `<SQL procedure statement>` to access the data.

The optional `<triggered when clause>` is a search condition, similar to the search condition of a DELETE or UPDATE statement. If the search condition is not TRUE for a row, then the trigger is not activated for that row.

The `<SQL procedure statement>` is limited to INSERT, DELETE, UPDATE and MERGE statements.

The `<HSQLDB trigger class FQN>` is a delimited identifer that contains the fully qualified name of a Java class that implements the `org.hsqldb.Trigger` interface.

Early releases of HyperSQL version 2.0 do not allow the use of OLD TABLE or NEW TABLE in statement level triggers.

TRIGGERED SQL STATEMENT

triggered SQL statement

The `<triggered SQL statement>` has three forms.

The first form is a single SQL procedure statement. This statement can reference the OLD ROW and NEW ROW variables. For example, it can reference these variables and insert a row into a separate table.

The second form is enclosed in a BEGIN ... END block and can include one or more SQL procedure statements. In BEFORE triggers, you can include SET statements to modify the inserted or updated rows. In AFTER triggers, you can include INSERT, DELETE and UPDATE statements to change the data in other database tables. SELECT and CALL statements are allowed in BEFORE and AFTER triggers. CALL statements in BEFORE triggers should not modify data.

The third form specifies a call to a Java method.

An example of a trigger with a block is given below. The block can include elements discussed the SQL-Invoked Routines chapter, including local variables, loops and conditionals. You can also raise an exception in such blocks in order to terminate the execution of the SQL statement that caused the trigger to execute.

```
create trigger trig after insert on testtrig
   referencing new row as newrow
   for each row when (newrow.id > 1)
   begin atomic
   insert into triglog values (newrow.id, newrow.data, 'inserted');
   /* more statements can be included */
   end
```

TRIGGER EXECUTION ORDER

trigger execution order

```
<trigger execution order> ::= BEFORE <other trigger name>
```

HyperSQL extends the SQL Standard to allow the order of execution of a trigger to be specified by using [BEFORE <other trigger name>] in the definition. The newly defined trigger will be executed before the specified other trigger. If this clause is not used, the new trigger is executed after all the previously defined triggers of the same scope (BEFORE, AFTER, EACH ROW, EACH STATEMENT).

DROP TRIGGER

drop trigger statement

```
<drop trigger statement> ::= DROP TRIGGER <trigger name>
```

Destroy a trigger.

Chapter 10. Built In Functions

Fred Toussi, The HSQL Development Group

$Revision: 3601 $

Published $Date: 2010-05-31 20:17:47 -0400 (Mon, 31 May 2010) $

Overview

HyperSQL supports a wide range of built-in functions and allows user-defined functions written in SQL and Java languages. User defined functions are covered in a separate chapter. If a built-in function is not available, you can write your own using SQL. Aggregate functions are discussed in chapters that cover SQL in general.

The built-in functions fall into three groups:

- SQL Standard Functions

  A wide rang of functions defined by SQL/Foundation are supported. SQL/Foundation functions that have no parameter are called without empty parentheses. Functions with multiple parameters often use keywords instead of commas to separate the parameters. Many functions are overloaded. Among these, some have one or more optional parameters that can be omitted, while the return type of some functions is dependent upon the type of one of the parameters. The usage of SQL Standard Functions (where they can be used) is covered more extensively in the Data Access and Change chapter

- JDBC Open Group CLI Functions

  These functions were defined as an extension to the CLI standard, which is the basis for ODBC and JDBC and supported by many database products. JDBC supports an escape mechanism to specify function calls in SQL statements in a manner that is independent of the function names supported by the target database engine. For example `SELECT {fn DAYOFMONTH (dateColumn)} FROM myTable` can be used in JDBC and is translated to Standard SQL as `SELECT EXTRACT (DAY_OF_MONTH FROM dateColumn) FROM myTable` if a database engine supports the Standard syntax. If a database engine does not support Standard SQL, then the translation will be different. HyperSQL supports all the function names specified in the JDBC specifications as native functions. Therefore, there is no need to use the `{fn FUNC_NAME ( ... ) }` escape with HyperSQL. If a JDBC function is supported by the SQL Standard in a different form, the SQL Standard form is the preferred form to use.

- HyperSQL Built-In Functions

  Several additional built-in functions are available for some useful operations. Some of these functions return the current setting for the session and the database. The General Functions accept arguments of different types and return values based on comparison between the arguments.

In the BNF specification used here, words in capital letters are actual tokens. Syntactic elements such as expressions are enclosed in angle brackets. The `<left paren>` and `<right paren>` tokens are represented with the actual symbol. Optional elements are enclosed with square brackets (`<left bracket>` and `<right bracket>`). Multiple options for a required element are enclosed with braces (`<left brace>` and `<right brace>`). Alternative tokens are separated with the vertical bar (`<vertical bar>`). At the end of each function definition, the standard which specifies the function is noted in parentheses as JDBC or HyperSQL, unless the function is in the SQL/Foundation part of the SQL Standard.

String and Binary String Functions

In SQL, there are three kinds of string: character, binary and bit. The units are respectively characters, octets, and bits. Each kind of string can be in different data types. CHAR, VARCHAR and CLOB are the character data types. BINARY, VARBINARY and BLOB are the binary data types. BIT and BIT VARYING are the bit string types. In all string functions, the position of a unit of the string within the whole string is specified from 1 to the length of the whole string. In the BNF, `<char value expr>` indicates any valid SQL expression that evaluates to a character type. Likewise, `<binary value expr>` indicates a binary type and `<num value expr>` indicates a numeric type.

ASCII

```
ASCII ( <char value expr> )
```

Returns an INTEGER equal to the ASCII code value of the first character of `<char value expr>`. (JDBC)

```
CHAR ( <UNICODE code> )
```

The argument is an INTEGER. Returns a character string containing a single character that has the specified `<UNI-CODE code>`, which is an integer. ASCII codes are a subset of the allowed values for `<UNICODE code>`. (JDBC)

CONCAT

```
CONCAT ( <char value expr 1>, <char value expr 2> )

CONCAT ( <binary value expr 1>, <binary value expr 2> )
```

The arguments are character strings or binary strings. Returns a string formed by concatenation of the arguments. Equivalent to the SQL concatenation expression `<value expr 1> || <value expr 2>`. (JDBC)

DIFFERENCE

```
DIFFERENCE ( <char value expr 1>, <char value expr 2> )
```

The arguments are character strings. Converts the arguments into SOUNDEX codes, and returns an INTEGER between 0-4 which indicates how similar the two SOUNDEX value are. If the values are the same, it returns 4, if the values have no similarity, it returns 0. In-between values are returned for partial similarity. (JDBC)

INSERT

```
INSERT ( <char value expr 1>, <offset>, <length>, <char value expr 2> )
```

Returns a character string based on `<char value expr 1>` in which `<length>` characters have been removed from the `<offset>` position and in their place, the whole `<char value expr 2>` is copied. Equivalent to SQL/Foundation `OVERLAY( <char value expr1> PLACING < char value expr2> FROM <offset> FOR <length> )`. (JDBC)

HEXTORAW

```
HEXTORAW( <char value expr> )
```

Returns a BINARY string formed by translation of hexadecimal digits and letters in the `<char value expr>`. Each character of the `<char value expr>` must be a digit or a letter in the A | B | C | D | E | F set. Each byte of the retired binary string is formed by translating two hex digits into one byte. (HyperSQL)

LCASE

```
LCASE ( <char value expr> )
```

Returns a character string that is the lower case version of the `<char value expr>`. Equivalent to SQL/Foundation `LOWER (<char value expr>)`. (JDBC)

LEFT

`LEFT ( <char value expr>, <length> )`

Returns a character string consisting of the first `<length>` characters of `<char value expr>`. Equivalent to SQL/Foundation `SUBSTRING(<char value expr> FROM 0 FOR <length>)`. (JDBC)

LENGTH

`LENGTH ( <char value expr> )`

Returns as a BIGINT value the number of characters in `<char value expr>`. Equivalent to SQL/Foundation `CHAR_LENGTH(<char value expr>)`. (JDBC)

LOCATE

`LOCATE ( <char value expr 1>, <char value expr 2> [ , <offset> ] )`

Returns as a BIGINT value the starting position of the first occurrence of `<char value expr 1>` within `<char value expr 2>`. If `<offset>` is specified, the search begins with the position indicated by `<offset>`. If the search is not successful, 0 is returned. Equivalent to SQL/Foundation `POSITION(<char value expr 1> IN <char value expr 2>)`. (JDBC)

LTRIM

`LTRIM ( <char value expr> )`

Returns a character string based on `<char value expr>` with the leading space characters removed. Equivalent to SQL/Foundation `TRIM( LEADING ' ' FROM <char value expr> )`. (JDBC)

RAWTOHEX

`RAWTOHEX( <binary value expr> )`

Returns a character string composed of hexadecimal digits representing the bytes in the `<binary value expr>`. Each byte of the `<binary value expr>` is translated into two hex digits. (HyperSQL)

REGEXP_MATCHES

`REGEXP_MATCHES ( <char value expr>, <regular expression> )`

Returns true if the <char value expr> matches the <regular expression>. The <regular expression> is defined according to Java language rules. (HyperSQL)

REPEAT

`REPEAT ( <char value expr>, <count> )`

Returns a character string based on `<char value expr>`, repeated `<count>` times. (JDBC)

REPLACE

`REPLACE ( <char value expr 1>, <char value expr 2>, <char value expr 3> )`

Returns a character string based on `<char value expr 1>` where each occurrence of `<char value expr 2>` has been replaced with a copy of `<char value expr 3>`. (JDBC)

REVERSE

REVERSE (<char value expr>)

Returns a character string based on <char value expr> with characters in the reverse order. (HyperSQL)

RIGHT

RIGHT (<char value expr>, <count>)

Returns a character string consisting of the last <count> characters of <char value expr>. (JDBC)

RTRIM

RTRIM (<char value expr>)

Returns a character string based on <char value expr> with the trailing space characters removed. Equivalent to SQL/Foundation TRIM(TRAILING ' ' FROM <character string>). (JDBC)

SOUNDEX

SOUNDEX (<char value expr>)

Returns a four character code representing the sound of <char value expr>. The US census algorithm is used. For example the soundex value for Washington is W252. (JDBC)

SPACE

SPACE (<count>)

Returns a character string consisting of <count> spaces. (JDBC)

SUBSTR

{ SUBSTR | SUBSTRING } (<char value expr>, <offset>, <length>)

The JDBC version of SQL/Foundation SUBSTRING returns a character string that consists of <length> characters from <char value expr> starting at the <offset> position. (JDBC)

UCASE

UCASE (<char value expr>)

Returns a character string that is the lower case version of the <char value expr>. Equivalent to SQL/Foundation UPPER(<char value expr>). (JDBC)

CHARACTER_LENGTH

{ CHAR_LENGTH | CHARACTER_LENGTH } (<char value expression> [USING { CHARACTERS | OCTETS }])

OCTET_LENGTH

OCTET_LENGTH (<string value expression>)

BIT_LENGTH

BIT_LENGTH (<string value expression>)

The CHAR_LENGTH or CHARACTER_LENGTH function can be used with character strings, while OCTET_LENGTH can be used with character or binary strings and BIT_LENGTH can be used with character, binary and bit strings.

All functions return a BIGINT value that measures the length of the string in the given unit. CHAR_LENGTH counts characters, OCTET_LENGTH counts octets and BIT_LENGTH counts bits in the string. For CHAR_LENGTH, if `[ USING OCTETS ]` is specified, the octet count is returned. (Foundation)

OVERLAY

```
OVERLAY ( <char value expr 1> PLACING <char value expr 2>

FROM <start position> [ FOR <string length> ] [ USING CHARACTERS ] )

OVERLAY ( <binary value expr 1> PLACING <binary value expr 2>

FROM <start position> [ FOR <string length> ] )
```

The character version of OVERLAY returns a character string based on `<char value expr 1>` in which `<string length>` characters have been removed from the `<start position>` and in their place, the whole `<char value expr 2>` is copied.

The binary version of OVERLAY returns a binary string formed in the same manner as the character version. (Foundation)

POSITION

```
POSITION ( <char value expr 1> IN <char value expr 2> [ USING CHARACTERS ] )

POSITION ( <binary value expr 1> IN <binary value expr 2> )
```

The character and binary versions of POSITION search the string value of the second argument for the first occurrence of the first argument string. If the search is successful, the position in the string is returned as a BIGINT. Otherwise zero is returned.

SUBSTRING

```
SUBSTRING ( <char value expr> FROM <start position> [ FOR <string length> ]
[ USING CHARACTERS ] )

SUBSTRING ( <binary value expr> FROM <start position> [ FOR <string length> ] )
```

The character version of SUBSTRING returns a character string that consists of the characters of the `<char value expr>` from `<start position>`. If the optional `<string length>` is specified, only `<string length>` characters are returned.

The binary version of SUBSTRING returns a binary string in the same manner. (Foundation)

TRIM

```
TRIM ([ [ LEADING | TRAILING | BOTH ] [ <trim character> ] FROM ] <char value
expr> )

TRIM ([ [ LEADING | TRAILING | BOTH ] [ <trim octet> ] FROM ] <binary value expr> )
```

The character version of TRIM returns a character string based on `<char value expr>`. Consecutive instances of `<trim character>` are removed from the beginning, the end or both ends of the `<char value expr>` depending on the value of the optional first qualifier `[ LEADING | TRAILING | BOTH ]`. If no qualifier

is specified, `BOTH` is used as default. If [`<trim character>`] is not specified, the space character is used as default.

The binary version of TRIM returns a binary string based on `<binary value expr>`. Consecutive instances of `<trim octet>` are removed in the same manner as in the character version. If [`<trim octet>`] is not specified, the 0 octet is used as default. (Foundation)

Numeric Functions

ABS

`ABS ( <num value expr> | <interval value expr> )`

Returns the absolute value of the argument as a value of the same type. (JDBC and Foundation)

ACOS

`ACOS ( <num value expr> )`

Returns the arc-cosine of the argument in radians as a value of DOUBLE type. (JDBC)

ASIN

`ASIN ( <num value expr> )`

Returns the arc-sine of the argument in radians as a value of DOUBLE type. (JDBC)

ATAN

`ATAN ( <num value expr> )`

Returns the arc-tangent of the argument in radians as a value of DOUBLE type. (JDBC)

ATAN2

`ATAN2 ( <num value expr 1>, <num value expr 2> )`

The `<num value expr 1>` and `<num value expr 2>` express the x and y coordinates of a point. Returns the angle, in radians, representing the angle coordinate of the point in polar coordinates, as a value of DOUBLE type. (JDBC)

CEILING

`{ CEIL | CEILING } ( <num value expr> )`

Returns the smallest integer greater than or equal to the argument. If the argument is exact numeric then the result is exact numeric with a scale of 0. If the argument is approximate numeric, then the result is of DOUBLE type. (JDBC and Foundation)

BITAND

`BITAND ( <num value expr 1>, <num value expr 2> )`

`BITAND ( <bit value expr 1>, <bit value expr 2> )`

BITOR

`BITOR ( <num value expr 1>, <num value expr 2> )`

```
BITOR ( <bit value expr 1>, <bit value expr 2> )
```

BITXOR

```
BITXOR ( <num value expr 1>, <num value expr 2> )
```

```
BITXOR ( <bit value expr 1>, <bit value expr 2> )
```

These three functions perform the bit operations: OR, AND, XOR, on two values. The values are either integer values, or bit strings. The result is an integer value of the same type as the arguments, or a bit string of the same length as the argument. Each bit of the result is formed by performing the operation on corresponding bits of the arguments. (HyperSQL)

COS

```
COS ( <num value expr> )
```

Returns the cosine of the argument (an angle expressed in radians) as a value of DOUBLE type. (JDBC)

COT

```
COT ( <num value expr> )
```

Returns the cotangent of the argument as a value of DOUBLE type. The `<num value expr>` represents an angle expressed in radians. (JDBC)

DEGREES

```
DEGREES ( <num value expr> )
```

Converts the argument (an angle expressed in `radians`) into degrees and returns the value in the DOUBLE type. (JDBC)

EXP

```
EXP ( <num value expr> )
```

Returns the exponential value of the argument as a value of DOUBLE type. (JDBC and Foundation)

FLOOR

```
FLOOR ( <num value expr> )
```

Returns the largest integer that is less than or equal to the argument. If the argument is exact numeric then the result is exact numeric with a scale of 0. If the argument is approximate numeric, then the result is of DOUBLE type. (JDBC and Foundation)

LN

```
LN ( <num value expr> )
```

Returns the natural logarithm of the argument, as a value of DOUBLE type. (Foundation)

LOG

```
LOG ( <num value expr> )
```

Returns the natural logarithm of the argument, as a value of DOUBLE type. (JDBC)

LOG10

```
LOG10 ( <num value expr> )
```

Returns the base 10 logarithm of the argument as a value of DOUBLE type. (JDBC)

```
MOD ( <num value expr 1>, <num value expr 2> )
```

MOD

Returns the remainder (modulus) of `<num value expr 1>` divided by `<num value expr 2>.` The data type of the returned value is the same as the second argument. (JDBC and Foundation)

PI

```
PI ()
```

Returns the constant pi as a value of DOUBLE type. (JDBC)

POWER

```
POWER ( <num value expr 1>, <num value expr 2> )
```

Returns the value of `<num value expr 1>` raised to the power of `<int value expr 2>` as a value of DOUBLE type. (JDBC and Foundation)

RADIANS

```
RADIANS ( <num value expr> )
```

Converts the argument (an angle expressed in `degrees`) into radians and returns the value in the DOUBLE type. (JDBC)

RAND

```
RAND ( [ <int value expr> ] )
```

Returns a random value in the DOUBLE type. The optional `[ <int value expr> ]` is used as seed value. In HyperSQL each session has a separate random number generator. The first call that uses a seed parameter sets the seed for subsequent calls that do not include a parameter. (JDBC)

ROUND

```
ROUND ( <num value expr>, <int value expr> )
```

The `<num value expr>` is of the DOUBLE type. The function returns a DOUBLE value which is the value of the argument rounded to `<int value expr>` places right of the decimal point. If `<int value expr>` is negative, the first argument is rounded to `<int value expr>` places to the left of the decimal point. (JDBC)

SIGN

```
SIGN ( <num value expr> )
```

Returns an INTEGER, indicating the sign of the argument. If the argument is negative then -1 is returned. If it is equal to zero then 0 is returned. If the argument is positive then 1 is returned. (JDBC)

SIN

```
SIN ( <num value expr> )
```

Returns the sine of the argument (an angle expressed in radians) as a value of DOUBLE type. (JDBC)

SQRT

```
SQRT ( <num value expr> )
```

Returns the square root of the argument as a value of DOUBLE type. (JDBC and Foundation)

TAN

```
TAN ( <num value expr> )
```

Returns the tangent of the argument (an angle expressed in radians) as a value of DOUBLE type. (JDBC)

TRUNCATE

```
TRUNCATE ( <num value expr>, <int value expr> )
```

Returns a value in the same type as `<num value expr>`. The value is rounded by replacing digits with zeros from `<int value expr>` places right of the decimal point to the end. If `<int value expr>` is negative, `ABS( <int value expr> )` digits to left of the decimal point and all digits to the right of the decimal points are replaced with zeros. Results of calling TRUNCATE with 12345.6789 with (-2, 0, 2, 4) are (12300.0000, 12345.0000, 12345.6700, 12345.6789). (JDBC)

Date Time and Interval Functions

TIMEZONE

```
TIMEZONE()
```

Returns the current time zone for the session. Returns an INTERVAL HOUR TO MINUTE value. (HyperSQL)

SESSION_TIMEZONE

```
SESSION_TIMEZONE()
```

Returns the default time zone for the current session. Returns an INTERVAL HOUR TO MINUTE value. (HyperSQL)

DATABASE_TIMEZONE

```
DATABASE_TIMEZONE()
```

Returns the time zone for the database engine. This is based on where the database server process is located. Returns an INTERVAL HOUR TO MINUTE value. (HyperSQL)

EXTRACT

```
EXTRACT ( <extract field> FROM <extract source> )

<extract field> ::= YEAR | MONTH | DAY | HOUR | MINUTE | DAY_OF_WEEK | WEEK_OF_YEAR
| QUARTER | DAY_OF_YEAR | DAY_OF_MONTH |

TIMEZONE_HOUR | TIMEZONE_MINUTE | SECOND | SECONDS_SINCE_MIDNIGHT |

DAY_NAME | MONTH_NAME
```

```
<extract source> ::= <datatime value expr> | <interval value expr>
```

The EXTRACT function returns a field or element of the `<extract source>`. The `<extract source>` is a datetime or interval expression. The type of the return value is BIGINT for most of the `<extract field>` options. The exceptions is `SECOND` where a DECIMAL value is returned which has the same precision as the datetime or interval expression. The field values `DAY_NAME` or `MONTH_NAME` result in a character string. When `MONTH_NAME` is specified, a string in the range January - December is returned. When `DAY_NAME` is specified, a string in the range Sunday -Saturday is returned.

If the `<extract source>` is FROM `<datatime value expr>`, different groups of `<extract source>` can be used depending on the data type of the expression. The `TIMEZONE_HOUR` | `TIMEZONE_MINUTE` options are valid only for TIME WITH TIMEZONE and TIMESTAMP WITH TIMEZONE data types. The `HOUR` | `MINUTE` | `SECOND` | `SECONDS_MIDNIGHT` options, are valid for TIME and TIMESTAMP types. The rest of the fields are valid for DATE and TIMESTAMP types.

If the `<extract source>` is FROM `<interval value expr>`, the `<extract field>` must be one of the fields of the INTERVAL type of the expressions. The `YEAR` | `MONTH` options may be valid for INTERVAL types based on months. The `DAY` | `HOUR` | `MINUTE` | `SECOND` | `SECONDS_MIDNIGHT` options may be valid for INTERVAL types based on seconds. For example, `DAY` | `HOUR` | `MINUTE` are the only valid fields for the INTERVAL DAY TO MINUTE data type. (Foundation with HyperSQL extensions)

CURRENT_DATE

```
CURRENT_DATE
```

CURRENT_TIME

```
CURRENT_TIME [ ( <time precision> ) ]
```

LOCALTIME

```
LOCALTIME [ ( <time precision> ) ]
```

CURRENT_TIMESTAMP

```
CURRENT_TIMESTAMP [ ( <timestamp precision> ) ]
```

LOCALTIMESTAMP

```
LOCALTIMESTAMP [ ( <timestamp precision> ) ]
```

These datetime functions return the datetime value representing the moment the function is called. CURRENT_DATE returns a value of DATE type. CURRENT_TIME returns a value of TIME WITH TIME ZONE type. LOCALTIME returns a value of TIME type. CURRENT_TIMESTAMP returns a value of TIMESTAMP WITH TIME ZONE type. LOCALTIMESTAMP returns a value of TIMESTAMP type. If the optional [(`<time precision>`)] or [(`<timestamp precision>`)] is used, then the returned value has the specified fraction of the second precision. (Foundation)

CURDATE

```
CURDATE ()
```

This function is equivalent to `CURRENT_DATE.` (JDBC)

CURTIME

```
CURTIME ()
```

This function is equivalent to `LOCALTIME`. (JDBC)

DAYNAME

`DAYNAME ( <datatime value expr> )`

This function is equivalent to `EXTRACT ( DAY_NAME FROM ... )` Returns a string in the range of Sunday - Saturday. (JDBC)

DAYOFMONTH

`DAYOFMONTH ( <datetime value expr> )`

This function is equivalent to `EXTRACT ( DAY_OF_MONTH FROM ... )` Returns an integer value in the range of 1-31. (JDBC)

DAYOFWEEK

`DAYOFWEEK ( <datetime value expr> )`

This function is equivalent to `EXTRACT ( DAY_OF_WEEK FROM ... )` Returns an integer value in the range of 1-7. The first day of the week is Sunday. (JDBC)

DAYOFYEAR

`DAYOFYEAR ( <datetime value expr> )`

This function is equivalent to `EXTRACT ( DAY_OF_YEAR FROM ... )` Returns an integer value in the range of 1-366. (JDBC)

HOUR

`HOUR ( <datetime value expr> )`

This function is equivalent to `EXTRACT ( HOUR FROM ... )` Returns an integer value in the range of 0-23. (JDBC)

MINUTE

`MINUTE ( <datetime value expr> )`

This function is equivalent to `EXTRACT ( MINUTE FROM ... )` Returns an integer value in the range of 0 - 59. (JDBC)

MONTH

`MONTH ( <datetime value expr> )`

This function is equivalent to `EXTRACT ( MONTH FROM ... )` Returns an integer value in the range of 1-12. (JDBC)

MONTHNAME

`MONTHNAME ( <datetime value expr> )`

This function is equivalent to `EXTRACT ( NAME_OF_MONTH FROM ... )` Returns a string in the range of January - December. (JDBC)

NOW

```
NOW ()
```

This function is equivalent to `LOCAL_TIMESTAMP`.

QUARTER

```
QUARTER ( <datetime value expr> )
```

This function is equivalent to `EXTRACT ( QUARTER FROM ... )` Returns an integer in the range of 1 - 4. (JDBC)

SECOND

```
SECOND ( <datetime value expr> )
```

This function is equivalent to `EXTRACT ( SECOND FROM ... )` Returns an integer or decimal in the range of 0 - 59, with the same precision as the <datetime value expr>. (JDBC)

SECONDS_SINCE_MIDNIGHT

```
SECONDS_SINCE_MIDNIGHT ( <datetime value expr> )
```

This function is equivalent to `EXTRACT ( SECONDS_SINCE_MIDNIGHT FROM ... )` Returns an integer in the range of 0 - 86399. (HyperSQL)

WEEK

```
WEEK ( <datetime value expr> )
```

This function is equivalent to `EXTRACT ( WEEK_OF_YEAR FROM ... )` Returns an integer in the range of 1 - 54. (JDBC)

YEAR

```
YEAR ( <datetime value expr> )
```

This function is equivalent to `EXTRACT ( YEAR FROM ... )` Returns an integer in the range of 1 - 9999. (JDBC)

TIMESTAMPADD

```
TIMESTAMPADD ( <tsi datetime field>, <numeric value expression>, <datetime value
expr>)
```

TIMESTAMPDIFF

```
TIMESTAMPDIFF ( <tsi datetime field>, <datetime value expr 1>, <datetime value
expr 2>)
```

```
<tsi datetime field> ::= SQL_TSI_FRAC_SECOND | SQL_TSI_SECOND | SQL_TSI_MINUTE
| SQL_TSI_HOUR | SQL_TSI_DAY | SQL_TSI_WEEK | SQL_TSI_MONTH | SQL_TSI_QUARTER
| SQL_TSI_YEAR
```

HyperSQL supports full SQL Standard datetime features. It supports adding integers representing units of time directly to datetime values using the arithmetic plus operator. It also supports subtracting one <datetime value expr> from another in the given units of days using the minus operator. An example of <datetime value expr> + <numeric value expression> <datetime field> is `LOCAL_TIMESTAMP + 5 DAY`. An example

of (`<datetime value expr>` - `<numeric value expression>`) `<datetime field>` is (`CURRENT_DATE` - `DATE '2008-08-8'`) `MONTH` which returns the number of calendar months between the two dates.

The two JDBC functions, `TIMESTAMPADD` and `TIMESTAMPDIFF` perform the same function as above SQL expressions. The field names are keywords and are different from those used in the EXTRACT functions. These names are valid for use only when calling these two functions. The return value for TIMESTAMPADD is of the same type as the datetime argument used. The return type for TIMESTAMPDIFF is always BIGINT, regardless of the type of arguments. The two datetime arguments of TIMESTAMPDIFF should be of the same type. (JDBC)

DATEADD

`DATEADD ( <field>, <numeric value expr>, <datetime value expr> )`

DATEDIFF

`DATEDIFF ( <field>, <datetime value expr 1>, datetime value expr 2> )`

`<field> ::= 'yy' | 'mm' | 'dd' | 'hh' | 'mi' | 'ss' | 'ms'`

The DATEADD and DATEDIFF functions are alternatives to TIMESTAMPADD and TIMESTAMPDIFF, with fewer available field options. The field names are specified as strings, rather than keywords. The fields translate to YEAR, MONTH, DAY, HOUR, MINUTE, SECOND and MILLISECOND. (HyperSQL}

TO_CHAR

`TO_CHAR( <datetime value expr>, <char value expr> )`

This function formats a datetime or numeric value to the format specified by the pattern given in the second argument. The pattern can contain pattern elements from the list given below, plus punctuation and space characters. An example, including the result, is given below:

```
TO_CHAR ( TIMESTAMP'2008-02-01 20:30:40', 'YYYY BC MONTH, DAY HH')

2008 AD February, Friday 8
```

The format is internally translated to a `java.text.SimpleDateFormat` format string. Any character sequences not listed below are included in the Java format string and may cause unexpected results or errors. Therefore unsupported format strings should not be used. The supported format components are as follows:

Table 10.1. TO CHAR Values

`BC | B.C. | AD | A.D.`	Returns `AD` for common era and `BC` for before common era
`RRRR`	4-digit year
`YYYY`	4-digit year
`IYYY`	4-digit year
`YY`	2 digit year
`IY`	2 digit year
`IYYY`	4-digit year
`MM`	Month (01-12)
`MON`	Short three-letter name of month
`MONTH`	Name of month

WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	Week of year (1-52 or 1-53) based on the ISO standard.
DAY	Name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DY	Short three-letter name of day.
HH	Hour of day (0-11).
HH12	Hour of day (0-11).
HH24	Hour of day (0-23).
MI	Minute (0-59).
SS	Second (0-59).
FF	Fractional seconds.

Array Functions

Array functions are specialised functions with ARRAY parameters.

CARDINALITY

```
CARDINALITY( <array value expr> )
```

Returns the element count for the given array argument. (Foundation)

MAX_CARDINALITY

```
MAX_CARDINALITY( <array value expr> )
```

Returns the maximum allowed element count for the given array argument. (Foundation)

TRIM_ARRAY

```
TRIM_ARRAY( <array value expr>, <num value expr> )
```

Returns a new array that contains the elements of the `<array value expr>` minus the number of elements specified by the `<num value expr>.` Elements are discarded from the end of the array. (Foundation)

General Functions

General functions can take different types of arguments. Some General Functions accept a variable number of arguments.

COALESCE

```
COALESCE( <value expr 1>, <value expr 2> [, ...] )
```

Returns `<value expr 1>` if it is not null, otherwise returns `<value expr 2>` if not null and so on. The type of both arguments must be comparable. (Foundation)

CONVERT

```
CONVERT ( <value expr> , <data type> )

<data type> ::= { SQL_BIGINT | SQL_BINARY | SQL_BIT |SQL_BLOB | SQL_BOOLEAN
| SQL_CHAR | SQL_CLOB | SQL_DATE | SQL_DECIMAL | SQL_DATALINK |SQL_DOUBLE |
SQL_FLOAT | SQL_INTEGER | SQL_LONGVARBINARY | SQL_LONGNVARCHAR | SQL_LONGVARCHAR
| SQL_NCHAR | SQL_NCLOB | SQL_NUMERIC | SQL_NVARCHAR | SQL_REAL | SQL_ROWID
| SQL_SQLXML | SQL_SMALLINT | SQL_TIME | SQL_TIMESTAMP | SQL_TINYINT |
SQL_VARBINARY | SQL_VARCHAR} [ ( <precision, length or scale parameters> ) ]
```

The CONVERT function is a JDBC escape function, equivalent to the SQL standard CAST expression. It converts the `<value expr>` into the given `<data type>` and returns the value. The `<data type>` options are synthetic names made by prefixing type names with `SQL_`. Some of the `<data type>` options represent valid SQL types, but some are based on non-standard type names, namely { `SQL_LONGNVARCHAR` | `SQL_LONGVARBINARY` | `SQL_LONGVARCHAR` | `SQL_TINYINT` }. None of the synthetic names can be used in any other context than the CONVERT function.

The definition of CONVERT in the JDBC Standard does not allow the precision, scale or length to be specified. This is required by the SQL standard for BINARY, BIT, BLOB, CHAR, CLOB, VARBINARY and VARCHAR types and is often needed for DECIMAL and NUMERIC. Therefore, HyperSQL allows the use of precision, scale or length for the type definition when they are valid for the type definition. HyperSQL also allows the use of real type names (without the `SQL_` prefix). (JDBC)

DECODE

```
DECODE( <value expr main>, <value expr match 1>, <value expr result 1> [....,]
[, <value expr default>] )
```

DECODE takes at least 3 arguments. The `<value expr main>` is compared with `<value expr match 1>` and if it matches, `<value expr result 1>` is returned. If there are additional pairs of `<value expr match n>` and `<value expr result n>`, comparison is repeated until a match is found the result is returned. If no match is found, the `<value expr default>` is returned if it is specified, otherwise NULL is returned. The type of the return value is a combination of the types of the `<value expr result ... >` arguments. (HyperSQL)

GREATEST

```
GREATEST( <value expr 1>, [<value expr ...>, ...] )
```

The GREATEST function takes one or more arguments. It compares the arguments with each other and returns the greatest argument. The return type is the combined type of the arguments. Arguments can be of any type, so long as they are comparable. (HyperSQL)

IFNULL

```
IFNULL( <value expr 1>, <value expr 2> )
```

Returns `<value expr 1>` if it is not null, otherwise returns `<value expr 2>`. The type of both arguments must be the same. Equivalent to SQL Standard `COALESCE(<value expr 1>, <value expr 2>)` function. (JDBC)

LEAST

```
LEAST( <value expr 1>, [<value expr ...>, ...] )
```

The LEAST function takes one or more arguments. It compares the arguments with each other and returns the smallest argument. The return type is the combined type of the arguments. Arguments can be of any type, so long as they are comparable. (HyperSQL)

NULLIF

```
NULLIF( <value expr 1>, <value expr 2> )
```

Returns `<value expr 1>` if it is not equal to `<value expr 2>`, otherwise returns null. The type of both arguments must be the same. This function is a shorthand for a specific CASE expression. (Foundation)

NVL

```
NVL( <value expr 1>, <value expr 2> )
```

Returns `<value expr 1>` if it is not null, otherwise returns `<value expr 2>`. The type of the return value is the combined type of the two value expressions. For example, if <value expr 1> is an INTEGER column and `<value expr 2>` is a DOUBLE constant, the return type is DOUBLE. This function is the same as IFNULL and COALESCE (HyperSQL)

System Functions

CRYPT_KEY

```
CRYPT_KEY( <value expr 1>, <value expr 2> )
```

Returns a binary string representation of a cryptography key for the given cipher and cyptography provider. The cipher specification is specified by `<value expr 1>` and the provider by `<value expr 2>`. To use the default provider, specify null for `<value expr 2>`. (HyperSQL)

IDENTITY

```
IDENTITY ()
```

Returns the last IDENTITY value inserted into a row by the current session. The statement, CALL IDENTITY() can be made after an INSERT statement that inserts a row into a table with an IDENTITY column. The CALL IDENTITY() statement returns the last IDENTITY value that was inserted into a table by the current session. Each session manages this function call separately and is not affected by inserts in other sessions. The statement can be executed as a direct statement or a prepared statement. (HyperSQL)

DATABASE

```
DATABASE ()
```

Returns the file name (without directory information) of the database. (JDBC)

DATABASE_VERSION

```
DATABASE_VERSION ()
```

Returns the full version string for the database engine. For example, 2.0.1. (JDBC)

USER

```
USER ()
```

Equivalent to the SQL function `CURRENT_USER`. (JDBC)

CURRENT_USER

```
CURRENT_USER
```

CURRENT_ROLE

```
CURRENT_ROLE
```

SESSION_USER

```
SESSION_USER
```

SYSTEM_USER

```
SYSTEM_USER
```

CURRENT_SCHEMA

```
CURRENT_SCHEMA
```

CURRENT_CATALOG

```
CURRENT_CATALOG
```

These functions return the named current session attribute. They are all SQL Standard functions.

The CURRENT_USER is the user that connected to the database, or a user subsequently set by the SET AUTHO-RIZATION statement.

SESSION_USER is the same as CURRENT_USER

SYSTEM_USER is the user that connected to the database. It is not changed with any command until the session is closed.

CURRENT_SCHEMA is default schema of the user, or a schema subsequently set by the SET SCHEMA command.

CURRENT_CATALOG is always the same within a given HyperSQL database and indicates the name of the catalog.

ISAUTOCOMMIT

```
ISAUTOCOMMIT()
```

Returns TRUE if the session is in autocommit mode. (HyperSQL)

ISREADONLYSESSION

```
ISREADONLYSESSION()
```

Returns TRUE if the session is in read only mode. (HyperSQL)

ISREADONLYDATABASE

```
ISREADONLYDATABASE()
```

Returns TRUE if the database is a read only database. (HyperSQL)

ISREADONLYDATABASEFILES

```
ISREADONLYDATABASEFILES()
```

Returns TRUE if the database is a read-only files database. In this kind of database, it is possible to modify the data, but the changes are not persisted to the database files. (HyperSQL)

ISOLATION_LEVEL

```
ISOLATION_LEVEL()
```

Returns the current transaction isolation level for the session. Returns either READ COMMITTED or SERIALIZ-ABLE as a string. (HyperSQL)

SESSION_ISOLATION_LEVEL

```
SESSION_ISOLATION_LEVEL()
```

Returns the default transaction isolation level for the current session. Returns either READ COMMITTED or SERI-ALIZABLE as a string. (HyperSQL)

DATABASE_ISOLATION_LEVEL

```
DATABASE_ISOLATION_LEVEL()
```

Returns the default transaction isolation level for the database. Returns either READ COMMITTED or SERIALIZ-ABLE as a string. (HyperSQL)

TRANSACTION_CONTROL

```
TRANSACTION_CONTROL()
```

Returns the current transaction model for the database. Returns LOCKS, MVLOCKS or MVCC as a string. (Hyper-SQL)

Chapter 11. System Management and Deployment Issues

Fred Toussi, The HSQL Development Group

$Revision: 3630 $

Copyright 2002-2010 Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2010-06-06 10:44:27 -0400 (Sun, 06 Jun 2010) $

Mode of Operation and Tables

HyperSQL has many modes of operation and features that allow it to be used in very different scenarios. Levels of memory usage, speed and accessibility by different applications are influenced by how HyperSQL is deployed.

Mode of Operation

The decision to run HyperSQL as a separate server process or as an *in-process* database should be based on the following:

- When HyperSQL is run as a server on a separate machine, it is isolated from hardware failures and crashes on the hosts running the application.

- When HyperSQL is run as a server on the same machine, it is isolated from application crashes and memory leaks.

- Server connections are slower than *in-process* connections due to the overhead of streaming the data for each JDBC call.

- You can reduce client/server traffic using SQL Stored procedures to reduce the number of JDBC execute calls.

- During development, it is better to use a Server with server.silent=false, which displays the statements sent to the server on the console window.

- To improve speed of execution for statements that are executed repeatedly, reuse a parameterized PreparedStatement for the lifetime of the connections.

Tables

TEXT tables are designed for special applications where the data has to be in an interchangeable format, such as CSV (comma separated values). TEXT tables should not be used for routine storage of data.

MEMORY tables and CACHED tables are generally used for data storage. The difference between the two is as follows:

- The data for all MEMORY tables is read from the *.script file when the database is started and stored in memory. In contrast the data for cached tables is not read into memory until the table is accessed. Furthermore, only part of the data for each CACHED table is held in memory, allowing tables with more data than can be held in memory.

- When the database is shutdown in the normal way, all the data for MEMORY tables is written out to the disk. In comparison, the data in CACHED tables that has changed is written out during operation and at shutdown.

- The size and capacity of the data cache for all the CACHED tables is configurable. This makes it possible to allow all the data in CACHED tables to be cached in memory. In this case, speed of access is good, but slightly slower than MEMORY tables.

- For normal applications it is recommended that MEMORY tables are used for small amounts of data, leaving CACHED tables for large data sets. For special applications in which speed is paramount and a large amount of free memory is available, MEMORY tables can be used for large tables as well.

Large Objects

HyperSQL 2.0 supports dedicated storage and access to BLOB and CLOB objects. These objects can have huge sizes. BLOB or CLOB is specified as the type of a column of the table. Afterwards, rows can be inserted into the table using a PreparedStatement for efficient transfer of large LOB data to the database. In *mem:* catalogs, CLOB and BLOB data is stored in memory. In *file:* catalogs, this data is stored in a single separate file which has the extension *.lobs. The size of this file can grow to huge, terabyte figures.

LOB data should be store in the database using a JDBC PreparedStatement object. The streaming methods send the LOB to the database in one operation as a binary or character stream. Inside the database, the disk space is allocated as needed and the data is saved as it is being received. LOB data should be retrieved from the database using a JDBC ResultSet method. When a streaming method is used to retrieve a LOB, it is retrieved in large chunks in a transparent manner. LOB data can also be stored by calling a JDBC method with String or byte[] argument, but these methods limit the size of the LOB that can be stored or retrieved.

LOB data is not duplicated in the database when a lob is copied from one table to another. The disk space is reused when a LOB is deleted and is not contained in any table.

By using a dedicated LOB store, HyperSQL achieves consistently high speeds (usually over 20MB / s) for both storage and retrieval of LOBs.

The LOB catalog is stored in the database as a memory table. Therefore the amount of JVM memory should be increased when more than tens of thousands of LOBs are stored in the database.

Deployment context

The files used for storing HyperSQL database data are all in the same directory. New files are always created and deleted by the database engine. Two simple principles must be observed:

- The Java process running HyperSQL must have full privileges on the directory where the files are stored. This include create and delete privileges.

- The file system must have enough spare room both for the 'permanent' and 'temporary' files. The default maximum size of the *.log file is 50MB. The *.data file can grow to up to 16GB (more if the default has been increased). The .backup file can be up to the size of the *.data file. The *.lobs file can grow to several terabytes. The temporary files created at the time of a SHUTDOWN can be equal in size to the *.script file and the .data file.

Readonly Databases

A *file:* catalog can be made readonly permanently, or it can be opened as readonly. To make the database readonly, the property, value pair, readonly=true can be added to the .properties file of the database.

It is also possible to open a normal database as readonly. For this, the property can be included in the URL of the first connection to the database.

There is another option which allows MEMORY tables to be writable, but without persisting the changes at SHUT-DOWN. This option is activated with the property, value pair, files_readonly= true, which can be added to the .prop-

erties file of the database, or included in the URL of the first connection to the database. This option is useful for running application tests which operate on a predefined dataset.

Memory and Disk Use

Memory used by the program can be thought of as two distinct pools: memory used for table data which is not released unless the data is deleted and memory that can be released or is released automatically, including memory used for caching, building result sets and other internal operations such as storing the information needed for a rollback a transaction.

Most JVM implementations allocate up to a maximum amount of memory (usually 64 MB by default). This amount is generally not adequate when large memory tables are used, or when the average size of rows in cached tables is larger than a few hundred bytes. The maximum amount of allocated memory can be set on the Java command line that is used for running HyperSQL. For example, with Sun JVM, parameter -Xmx256m increases the amount to 256 MB.

Table Memory Allocation

The memory used for a MEMORY table is the sum of memory used by each row. Each MEMORY table row is a Java object that has 2 int or reference variables. It contains an array of objects for the fields in the row. Each field is an object such as `Integer`, `Long`, `String`, etc. In addition each index on the table adds a node object to the row. Each node object has 6 int or reference variables. As a result, a table with just one column of type INTEGER will have four objects per row, with a total of 10 variables of 4 bytes each - currently taking up 80 bytes per row. Beyond this, each extra column in the table adds at least a few bytes to the size of each row.

Result Set Memory Allocation

By default, all the rows in the result set are built in memory, so very large result sets may not be possible to build. In server mode databases, by default, the result set memory is released from the server once the database server has returned the result set. *in-process* databases release the memory when the application program releases the `java.sql.ResultSet` object. Server modes require additional memory for returning result sets, as they convert the full result set into an array of bytes which is then transmitted to the client.

HyperSQL 2.0 supports disk-based result sets. The commands, `SET SESSION RESULT MEMORY ROWS <integer>` and `SET DATABASE DEFAULT RESULT MEMORY ROWS <integer>` specify a threshold for the number of rows. Results with row counts above the threshold are stored on disk. These settings also apply to temporary tables and subquery tables.

When the setFetchSize() method of the Statement interface is used to limit the number rows fetched, the whole result is held by the engine and is returned to the JDBC ResultSet in blocks of rows of the specified fetch size. Disk-based result sets slow down the database operations and should be used only when absolutely necessary, perhaps with result sets that are larger than tens of thousands of rows.

Temporary Memory Use During Operations

When UPDATE and DELETE queries are performed on CACHED tables, the full set of rows that are affected, including those affected due to ON UPDATE actions, is held in memory for the duration of the operation. This means it may not be possible to perform deletes or updates involving very large numbers of rows of CACHED tables. Such operations should be performed in smaller sets.

When transactions support is enabled with SET AUTOCOMMIT FALSE, lists of all insert, delete or update operations are stored in memory so that they can be undone when ROLLBACK is issued. For CACHED tables, only the transaction information is held in memory, not the actual rows that have changed. Transactions that span thousands of modification

to data will take up a lot of memory until the next COMMIT or ROLLBACK clears the list. Each row modification uses less than 100 bytes until COMMIT.

When subqueries or views are used in SELECT and other statements, transient tables are created and populated by the engine. If the `SET SESSION RESULT MEMORY ROWS <integer>` statement has been used, these transient tables are stored on disk when they are larger than the threshold.

Data Cache Memory Allocation

With CACHED tables, the data is stored on disk and only up to a maximum number of rows are held in memory at any time. The default is up to 50,000 rows. The SET FILES CACHE ROWS command or the hsqldb.cache_rows connection property can be set to alter this amount. As any random subset of the rows in any of the CACHED tables can be held in the cache, the amount of memory needed by cached rows can reach the sum of the rows containing the largest field data. For example if a table with 100,000 rows contains 40,000 rows with 1,000 bytes of data in each row and 60,000 rows with 100 bytes in each, the cache can grow to contain 50,000 of the smaller rows, but as explained further, only 10,000 or the large rows.

An additional property, hsqldb.cache_size is used in conjunction with the hsqldb.cache_rows property. This puts a limit in bytes on the total size of rows that are cached. The default values is 10,000KB. (This is the size of binary images of the rows and indexes. It translates to more actual memory, typically 2-4 times, used for the cache because the data is represented by Java objects.)

If memory is limited, the hsqldb.cache_rows or hsqldb.cache_size database properties can be reduced. In the example above, if the hsqldb.cache_size is reduced from 10,000 to 5,000, it will allow the number of cached rows to reach 50,000 small rows, but only 5,000 of the larger rows.

Data for CLOB and BLOB columns is not cached and does not affect the CACHED table memory cache.

The use of Java nio file access method also increases memory usage. Access with nio improves database update speed and is used by default for data files up to 256 MB. For minimal memory use, nio access should be disabled.

Object Pool Memory Allocation

HyperSQL uses a set of fast pools for immutable objects such as Integer, Long and short String objects that are stored in the database. In most circumstances, this reduces the memory footprint still further as fewer copies of the most frequently-used objects are kept in memory. The object pools are shared among all databases in the JVM. The size of each pool can be modified only by altering and recompiling the `org.hsqldb.store.ValuePool` class.

Lob Memory Usage

Access to lobs is always performed in chunks, so it is perfectly possible to store and access a CLOB or BLOB that is larger than the JVM memory allocation. Early versions of HyperSQL 2.0 use memory-based tables for the lob catalog (not the data). Therefore it is practical to store about 100,000 individual lobs in the database with the default JVM memory allocation. More lobs can be stored with larger JVM memory allocations. The realistic maximum number of lobs stored in the database is probably about a million. The actual total size of lobs is almost unlimited. We have tested with over 100 GB of lobs without any loss of performance.

Disk Space

With file: database, the engine uses the disk for storage of data and any change. For safely, the engine backs up the data internally during operation. Spare space, at least equal to the size of the .data and .script file is needed. The .lobs file is not backed up during operation.

Managing Database Connections

In all running modes (server or *in-process*) multiple connections to the database engine are supported. *in-process* (standalone) mode supports connections from the client in the same Java Virtual Machine, while server modes support connections over the network from several different clients.

Connection pooling software can be used to connect to the database but it is not generally necessary. Connection pools may be used for the following reasons.

- To allow new queries to be performed while a time-consuming query is being performed in the background. In HyperSQL, blocking depends on the transaction control model, the isolation level, and the current activity by other sessions.

- To limit the maximum number of simultaneous connections to the database for performance reasons. With HSQLDB this can be useful if your application is designed in a way that opens and closes connections for each small task. Also, the overall performance may be higher when fewer simultaneous connections are used. If you want to reduce the number of simultaneous sessions, you can use a connection pool with fewer pooled connections.

An application that is not both multi-threaded and transactional, such as an application for recording user login and logout actions, does not need more than one connection. The connection can stay open indefinitely and reopened only when it is dropped due to network problems.

When using an *in-process* database, when the last connection to the database is closed, the database still remains open. An explicit SHUTDOWN command, with or without an argument, is required to close the database. A connection property on the connection URL or in a properties object can be used to shutdown the database when the last connection is closed.

When using a server database (and to some extent, an *in-process* database), care must be taken to avoid creating and dropping JDBC Connections too frequently. Failure to observe this will result in poor performance when the application is under heavy load.

A common error made by users in load-test simulations is to use a single client machine to open and close thousands of connections to a HyperSQL server instance. The connection attempts will fail after a few thousand because of OS restrictions on opening sockets and the delay that is built into the OS in closing them.

Tweaking the Mode of Operation

Different modes of operation and settings are used for different purposes. Some scenarios are discussed below:

Application Development and Testing

For application unit testing you can use an all-in-memory, in-process database.

If the tests are all run in one process, then the contents of a *mem:* database survives between tests. To release the contents you can use the SHUTDOWN command (an SQL command). You can even use multiple *mem:* databases in your tests and SHUTDOWN each one separately.

If the tests are in different processes and you want to keep the data between the tests, the best solution is to use a Server instance that has a *mem:* database. After the tests are done, you can SHUTDOWN this database, which will shutdown the server.

The Server has an option that allows databases to be created as needed by making a connection (see the Listeners Chapter). This option is useful for testing, as your server is never shut down. Each time you connect to the *mem:* database that is served by the Server, the database is created if it does not exist (i.e. has been previously shut down).

If you do not want to run a Server instance, and you need persistence between tests in different processes, then you should use a *file:* database. You can use the `shutdown=true` connection property to ensure the database is persisted fully after the connections are closed. An alternative option is to use `hsqldb.write_delay=false` connection property, but this is slightly slower than the other option.

It has been reported that some data access frameworks do not close all their connection to the database after the tests. In such situations, you need to use zero WRITE DELAY if you want the data to persist at the end of the tests

You may actually want to use a *file:* database, or a server instance that serves a *file:* database in preference to a *mem:* database. As HyperSQL logs the DDL and DML statements in the .log file, this file can be used to check what is being sent to the database. Note that UPDATE statements are represented by a DELETE followed by an INSERT statement. Statements are written out when the connection commits. The write delay also has an effect on how soon the statements are written out.

Some types of tests start with a database that already contains the tables and data, and perform various operations on it during the tests. You can create and populate the initial database then set the property "files_read_only=true" in the .properties file of the database. The tests can then modify the database, but these modifications are not persisted after the tests have completed.

Embedded Databases in Desktop Applications

In this usage, the amount of data change is often limited and there is often a requirement to persist the data immediately. You can use the property `write_delay=false` to force a disk sync after each commit. Before the application is closed, you should perform the SHUTDOWN command to ensure the database is opened instantly when it is next opened.

Embedded Databases in Server Applications

This usage involves a server application, such as a web application, connecting to an embedded HyperSQL instance. In this usage, the database is often accessed heavily, therefore performance and latency is a consideration. If the database is updated heavily, the default value of the WRITE DELAY property (1 sec) is often enough, as it is assumed the server or the application does not go down frequently. If it is necessary, you can reduce the WRITE DELAY to a small value (20 ms) without impacting the update speed. If you reduce WRITE DELAY to zero, performance drops to the speed of disk file sync operation.

Alternatively, a server application can use an all-in-mem database instance for fast access, while sending the data changes to a persistent, disk based instance either periodically or in real time.

Embedding a Database Listener

Since you won't be able to access *in-process* database instances from other processes, you will often want to run a Listener in your server applications with embedded databases. You can do this by starting up a Server or WebServer instance programmatically, but you could also use the class `org.hsqldb.util.MainInvoker` to start up your application and a Server or WebServer without any programming.

Example 11.1. MainInvoker Example

```
java -cp path/to/your/app.jar:path/to/hsqldb.jar your.App "" org.hsqldb.server.Server
```

(Use ; instead of : to delimit classpath elements on Windows). Specify the same *in-process* JDBC URL to your app and in the `server.properties` file. You can then connect to the database from outside using a JDBC URL like `jdbc:hsqldb:hsql://hostname`.

This tactic can be used to run off-the-shelf server applications with an embedded HyperSQL Server, without doing any coding.

`MainInvoker` can be used to run any number of Java class main method invocations in a single JVM. See the API spec for `MainInvoker` for details on its usage.

Using HyperSQL Without Logging

All file database that are not readonly, write changes to the .log file. There are scenarios where writing to the .log file can be turned off to improve performance, especially with larger databases. For these applications you can set the property `hsqldb.log_data=false` to disable the recovery log and speed up data change performance. The equivalent SQL command is SET FILES LOG FALSE.

With this setting, no data is logged, but all the changes to cached tables are written to the .data file. To persist all the data changes up to date, you can use the CHECKPOINT command. If you perform SHUTDOWN, the data is also persisted correctly. If you do not use CHECKPOINT or SHUTDOWN. All the changes are lost and the database reverts to its original state when it is opened.

Your server applications can use a database as a temporary disk data cache which is not persisted past the lifetime of the application. For this usage, delete the database files when the application ends.

On some platforms, such as embedded devices which are reliable, this is also a useful option. Your application issues CHECKPOINT to save the changes made so far. This method of use reduces write operations on SSD devices. For this usage, the lock file should also be disabled with the connection property hsqldb.lock_file=false.

Server Databases

Running databases in a HyperSQL server is the best overall method of access. As the JVM process is separate from the application, this method is the most reliable as well as the most accessible method of running databases.

Upgrading Databases

Any database that is not produced with the release version of HyperSQL 2.0 must be upgraded to this version. Most catalogs created with 1.8.x can be upgraded simply by opening with HyperSQL 2. When this is not possible due to errors, the rest of the procedures below should be followed.

Once a database is upgraded to 2.0, it can no longer be used with previous versions of HyperSQL.

If your database has been created with version 1.7.x, first upgrade to version 1.8.1 and perform a SHUTDOWN COM-PACT with this version. You can then open and upgrade the database with version 2.0.

Upgrading From Older Versions

To upgrade from version 1.8.x with the default TEXT format script files, simply open the database with 2.0. If the version 1.8.x files have database script format set to BINARY or COMPRESSED (ZIPPED) you must issue the SET SCRIPTFORMAT TEXT and SHUTDOWN SCRIPT commands with the old version, then open with the new version of the engine. In most cases the upgrade is successful and complete.

It is strongly recommended to execute SHUTDOWN COMPACT after an automatic upgrade from previous versions.

If your database has been created with version 1.7.2 or 1.7.3, first upgrade to version 1.8.1 and perform a SHUTDOWN COMPACT with this version. You can then upgrade the database to version 2.0.

To upgrade from older version database files (1.7.1 and older) that contain CACHED tables, use the SCRIPT procedure below. In all versions of HyperSQL, the `SCRIPT 'filename'` command (used as an SQL statement) allows you to save a full record of your database, including database object definitions and data, to a file of your choice. You can export a script file using the old version of the database engine and open the script as a database with 2.0.

Procedure 11.1. Upgrade Using the SCRIPT Procedure for Very Old Versions

1. Open the original database in the old version of DatabaseManager

2. Issue the SCRIPT command, for example `SCRIPT 'newversion.script'` to create a script file containing a copy of the database.

3. SHUTDOWN this database.

4. Copy the original `*.properties` file into `newversion.properties` in the same directory as `newversion.script`

5. Try to open the new database `newversion` using DatabaseManager of version 1.8.1.

6. If there is any inconsistency in the data, the script line number is reported on the console and the opening process is aborted. Edit and correct any problems in the `newversion.script` before attempting to open again. Use the guidelines in the next section (Manual Changes to the `.script` File). Use a programming editor that is capable of handling very large files and does not wrap long lines of text.

Manual Changes to the *.script File

In HyperSQL 2.0 the full range of ALTER TABLE commands is available to change the data structures and their names. However, if an old database cannot be opened due to data inconsistencies, or it uses index or column names that are not compatible with 2.0, manual editing of the `*.script` file can be performed.

*   Version 2.0 does not accept duplicate names for indexes that were allowed before 1.7.2.

*   Version 2.0 does not accept some table or column names that are SQL reserved keywords without double quoting.

*   Version 2.0 is more strict with check conditions and default values.

Other manual changes are also possible. Note that the `*.script` file must be the result of a SHUTDOWN SCRIPT and must contain the full data for the database. The following changes can be applied so long as they do not affect the integrity of existing data.

*   Names of tables, columns and indexes can be changed. These changes must be consistent regarding foreign key constraints.

*   `CHECK`

    A check constraint can always be removed.

*   `NOT NULL`

    A not-null constraint can always be removed.

*   `PRIMARY KEY`

    A primary key constraint can be removed. It cannot be removed if there is a foreign key referencing the column(s).

*   `UNIQUE`

    A UNIQUE constraint can be removed if there is no foreign key referencing the column(s).

*   `FOREIGN KEY`

    A FOREIGN KEY constraint can always be removed.

- `COLUMN TYPES`

  Some changes to column types are possible. For example an INTEGER column can be changed to BIGINT.

After completing the changes and saving the modified `.script` file, you can open the database as normal.

Backward Compatibility Issues

HyperSQL 2.0 conforms to the SQL Standard better than previous versions and supports more features. For these reasons, there may be some compatibility issues when converting old database, or using applications that were written for version 1.8.x or earlier. Some of the potential issues are listed here.

- User names and passwords are case-sensitive. Check the .script file of a database for the correct case of user name and password and use this form in the connection properties or on connection URL.

- Check constraints must conform to the SQL Standard. A check constraint is rejected if it is not deterministic or retrospectively deterministic. When opening an old database, HyperSQL silently drops check constraints that no longer compile. See under check constraints for more detail about what is not allowed.

- Type declarations in column definition and in cast expressions must have the necessary size parameters.

- In connection with the above, an old database that did not have the `enforce_strict_size` property, is now converted to version 2.0 with the engine supplying the missing size parameters. For example, a VARCHAR column declaration that has no size, is given a 32K size. Check these sizes are adequate for your use, and change the column definition as necessary.

- Column names in a GROUP BY clause were previously resolved to the column label. They are now resolved to column name first, and if the name does not match, to the column label.

- If two or more tables in a join contain columns with the same name, the columns cannot be referenced in join and where conditions. Use table names before column names to qualify the references to such columns.

- Table definitions containing GENERATED BY DEFAULT AS IDENTITY but with no PRIMARY KEY do not automatically create a primary key. Database .script files made with 1.8 are fine, as the PRIMARY KEY clause is always included. But your application program may assume an automatic primary key is created.

- CREATE ALIAS is now obsolete. Use the new function definition syntax. The `org.hsqldb.Library` class no longer exists. You should use the SQL form of the old library functions. For example, use `LOG(x)` rather than the direct form, `"org.hsqldb.Library.log"(x)`.

- The names of some commands for changing database and session properties have changed. See the list of statements in this chapter.

Backing Up Database Catalogs

The database engine saves the files containing all the data in a file catalog when a shutdown takes place. It automatically recovers from an abnormal termination and preserves the data when the catalog is opened next time. In an ideal operating environment, where there is no OS crash, disk failure, bugs in code, etc. there would be no need regularly to backup a database. This is meant to say, the engine performs the routine shutdown procedure internally, therefore backing up catalogs is an insurance policy against all sorts of misadventure that are not under the control of the database engine.

The data for each catalog consists of up to 5 files in the same directory with the endings such as `*.properties`, `*.script`, etc., as detailed in previous chapters.

HyperSQL 2.0 includes commands to backup the database files into a single `.tar` or `.tar.gz` file archive. The backup can be performed by a command given in a JDBC session if the target database catalog is running, or on the command-line if the target catalog has been shutdown.

Making Online Backups

To back up a running catalog, obtain a JDBC connection and issue a `BACKUP DATABASE` command in SQL. In its most simple form, the command format below will backup the database as a single `.tar.gz` file to the given directory.

```
BACKUP DATABASE TO <directory name> BLOCKING
```

See the next section under Statements for details about the command and its options. See the sections below about restoring a backup.

Making Offline Backups

To back up an offline catalog, the catalog must be in shut down state. You will run a Java command like this

Example 11.2. Offline Backup Example

```
java -cp path/to/hsqldb.jar org.hsqldb.lib.tar.DbBackup --save  \
tar/path.tar db/base/path
```

where `tar/path.tar` is a file path to the `*.tar` or `*.tar.gz` file to be created, and `db/base/path` is the file path to the catalog file base name (in same fashion as in `server.database.*` settings and JDBC URLs with catalog type *file:*.

Examining Backups

You can list the contents of backup tar files with `DbBackup` on your operating system command line, or with any Pax-compliant tar or pax client (this includes GNU tar),

Example 11.3. Listing a Backup with DbBackup

```
java -cp path/to/hsqldb.jar org.hsqldb.lib.tar.DbBackup --list tar/path.tar
```

You can also give regular expressions at the end of the command line if you are only interested in some of the file entries in the backup. Note that these are real regular expressions, not shell globbing patterns, so you would use `.+script` to match entries ending in "script", not `*script`.

You can examine the contents of the backup in their entirety by restoring the backup, as explained in the following section, to a temporary directory.

Restoring a Backup

You use `DbBackup` on your operating system command line to restore a catalog from a backup.

Example 11.4. Restoring a Backup with DbBackup

```
java -cp path/to/hsqldb.jar org.hsqldb.lib.tar.DbBackup --extract  \
    tar/path.tar db/dir
```

where `tar/path.tar` is a file path to the *.tar or *.tar.gz file to be read, and `db/dir` is the target directory to extract the catalog files into. Note that `db/dir` specifies a directory path, without the catalog file base name. The files will be created with the names stored in the tar file (and which you can see as described in the preceding section).

Encrypted Databases

HyperSQL supports encrypted databases. Encryption services use the Java Cryptography Extensions (JCE) and uses the ciphers installed with the JRE. HyperSQL itself does not contain any cryptography code.

Three elements are involved in specifying the encryption method and key. A cipher, together with its configuration is identified by a string which includes the name of the cipher and optional parameters. A provider is the fully qualified class name of the cipher provider. A key is represented as a hexadecimal string.

Creating and Accessing an Encrypted Database

First, a key must be created for the desired cipher and configuration. This is done by calling the function CRYPT_KEY(<cipher spec>, <provider>). If the default provider (the built-in JVM ciphers) is used, then NULL should be specified as the provider. The CRYPT_KEY function returns a hexadecimal key. The function call can be made in any HyperSQL database, so long as the provider class is on the classpath. This key can be used to create a new encrypted database. Calls to this function always return different keys, based on a generated random values.

As an example, a call to CRYPT_KEY('Blowfish', null) returned the string, '604a6105889da65326bf35790a923932'. To create a new database, the URL below is used:

```
jdbc:hsqldb:file:<database
path>;crypt_key=604a6105889da65326bf35790a923932;crypt_type=blowfish
```

The third property name is crypt_provider. This is specified only when the provider is not the default provider.

HyperSQL works with any symmetric cipher that may be available from the JVM.

The files that are encrypted include the .script, .data, .backup and .log files. The .lobs file is not encrypted by default. The property crypt_lobs=true must be specified to encrypt the .lobs file.

Speed Considerations

General operations on an encrypted database are performed the same as with any database. However, some operations are significantly slower than with the equivalent cleartext database. With MEMORY tables, there is no difference to the speed of SELECT statements, but data change statements are slower. With CACHED tables, the speed of all statements is slower.

Security Considerations

Security considerations for encrypted databases have been discussed at length in HSQLDB discussion groups. Development team members have commented that encryption is not a panacea for all security needs. The following issues should be taken into account:

- Encrypted files are relatively safe in transport, but because databases contain many repeated values and words, especially known tokens such as CREATE, INSERT, etc., breaking the encryption of a database may be simpler than an unknown file.

- Only the files are encrypted, not the memory image. Poking into computer memory, while the database is open, will expose the contents of the database.

- HyperSQL is open source. Someone who has the key, can compile and use a modified version of the program that saves a full cleartext dump of an encrypted database

Therefore encryption is generally effective only when the users who have access to the crypt key are trusted.

Monitoring Database Operations

Database operations can be monitored at different levels using internal HyperSQL capabilities or add-ons.

Statement Level Monitoring

Statement level monitoring allows you to gather statistics about executed statements. HyperSQL is supported by the monitoring tool JAMon (Java Application Monitor). JAMon is currently developed as the SourceForge project, jamonapi.

JAMon works at the JDBC level. It can monitor and gather statistics on different types of executed statements or other JDBC calls.

Early versions of JAMon were developed with HSQLDB and had to be integrated into HSQLDB at code level. The latest versions can be added on as a proxy in a much simpler fashion.

Internal Event Monitoring

HyperSQL can log important internal events of the engine. These events occur during the operation of the engine, and are not always coupled with the exact type of statement being executed. Normal events such as opening and closing of files, or errors such as OutOfMemory conditions are examples of logged events.

HyperSQL supports two methods of logging. One method is specific to the individual database and is managed internally by HyperSQL. The other method is specific to JVM and is managed by a logging framework.

The internally-generated, individual log for the database can be enabled with the `SET DATABASE EVENT LOG LEVEL` statement, described in the next section. This method of logging is very useful for desktop application deployment, as it provides an ongoing record of database operations.

HyperSQL also supports log4J and JDK logging. The same event information that is passed to the internal log, is passed to external logging frameworks. These frameworks are configured outside HyperSQL. The log messages include the unique id of the database that generated the message, so it can be identified in a multi-database server context.

Server Operation Monitoring

A Server or WebServer instance can be started with the property server.silent=false. This causes all the connections and their executed statements to be printed to stdout as the statements are submitted to the server.

Statements

System level statements are listed in this section. Statements that begin with SET DATABASE or SET FILES are for properties that have an effect on the normal operation of HyperSQL. The effects of these statements are also discussed in different chapters.

SHUTDOWN

shutdown statement

```
<shutdown statement> ::= SHUTDOWN [IMMEDIATELY | COMPACT | SCRIPT]
```

Shutdown the database. If the optional qualifier is not used, a normal SHUTDOWN is performed. A normal SHUTDOWN ensures all data is saved correctly and the database opens without delay on next use.

SHUTDOWN IMMEDI-ATELY	Saves the *.log file and closes the database files. This is the quickest form of shut-down. This command should not be used as the routine method of closing the database, because when the database is accessed next time, it may take a long time to start.
SHUTDOWN COMPACT	This is similar to normal SHUTDOWN, but reduces the *.data file to its minimum size. It takes longer than normal SHUTDOWN.
SHUTDOWN SCRIPT	This is similar to SHUTDOWN COMPACT, but it does not rewrite the `*.data` and text table files. After SHUTDOWN SCRIPT, only the `*.script` and `*.proper-ties` files remain. At the next startup, these files are processed and the `*.data` and `*.backup` files are created. This command in effect performs part of the job of SHUTDOWN COMPACT, leaving the other part to be performed automatically at the next startup.
	This command produces a full script of the database which can be edited for special purposes prior to the next startup.

Only a user with the DBA role can execute this statement.

BACKUP DATABASE

backup database statement

```
<backup database statement> ::= BACKUP DATABASE TO <file path> {SCRIPT | [NOT]
COMPRESSED} BLOCKING
```

Backup the database to specified `<file path>` for archiving purposes.

The `<file path>` can be in two forms. If the `<file path>` ends with a forward slash, it specifies a directory. In this case, an automatic name for the archive is generated that includes the date, time and the base name of the database. The database is backed up to this archive file in the specified directory. If the `<file path>` does not end with a forward slash, it specifies a user-defined file name for the backup archive. The archive is in tar, gzip format depending on whether it is compressed or not.

The SCRIPT option is not currently supported. If SCRIPT is specified, the backup will consist of two files, a `*.prop-erties` file and a `*.script` file, which contain all the data and settings of the database. These files are not com-pressed.

If COMPRESSED or NOT COMPRESSED is specified, the backup consists of the current snapshot of database files. During backup, a CHECKPOINT command is silently executed.

The qualifier, BLOCKING, means all database operations are suspended during backup.

The HyperSQL jar also contains a program that creates an archive of an offline database. It also contains a program to expand an archive into database files. These programs are documented in this chapter under Backing up Database Catalogs.

Only a user with the DBA role can execute this statement.

CHECKPOINT

checkpoint statement

```
<checkpoint statement> ::= CHECKPOINT [DEFRAG]
```

Closes the database files, rewrites the script file, deletes the log file and opens the database. If `DEFRAG` is specified, also shrinks the `*.data` file to its minumum size. Only a user with the DBA role can execute this statement.

Only a user with the DBA role can execute this statement.

CRYPT_KEY

crypt_key function

```
<crypt_key function> ::= CRYPT_KEY ( <cipher spec>, <provider> )
```

The statement, `CALL CRYPT_KEY( <cipher spec>, <provider> )` returns a binary string representing a valid key for the giver cipher and provider. The `<provider>` argument is specified as NULL for the default provider.

SCRIPT

script statement

```
<script statement> ::= SCRIPT [<file name>]
```

Returns a script containing SQL statements that define the database, its users, and its schema objects. If `<file name>` is not specified, the statements are returned in a ResultSet, with each row containing an SQL statement. No data statements are included in this form. The optional file name is a single-quoted string. If `<file name>` is specified, then the script is written to the named file. In this case, all the data in all tables of the database is included in the script as INSERT statements.

Only a user with the DBA role can execute this statement.

SET DATABASE COLLATION

set database collation statement

```
<set database collation statement> ::= SET DATABASE COLLATION <collation name>
```

Each database can have its own collation. Sets the collation from the set of collations supported by HyperSQL. Once this command has been issued, the database can be opened in any JVM and will retain its collation. Only a user with the DBA role can execute this statement.

Only a user with the DBA role can execute this statement.

SET DATABASE DEFAULT TABLE TYPE

set database default table type statement

```
<set database default table type> ::= SET DATABASE DEFAULT TABLE TYPE { CACHED
| MEMORY }
```

Sets the type of table created when the next CREATE TABLE statement is executed. The default is MEMORY.

Only a user with the DBA role can execute this statement.

SET DATABASE DEFAULT RESULT MEMORY ROWS

set database default result memory rows statement

```
<set database default result memory rows> ::= SET DATABASE DEFAULT RESULT MEMORY
ROWS <unsigned integer literal>
```

Sets the maximum number of rows of each result set and other internal temporary table that is held in memory. This setting applies to all sessions. Individual sessions can change the value with the `SET SESSION RESULT MEMORY ROWS` command. The default is 0, meaning all result sets are held in memory.

Only a user with the DBA role can execute this statement.

SET DATABASE EVENT LOG LEVEL

set database event log level statement*

```
<set database event log level> ::= SET DATABASE EVENT LOG LEVEL { 0 | 1 | 2 }
```

Sets the amount of information logged in the internal, database-specific event log. Level 0 means no log. Level 1 means only important (error) events. Level 2 means more events, including both important and less important (normal) events. For readonly and *mem:* databases, if the level is set above 0, the log messages are directed to stderr.

Only a user with the DBA role can execute this statement.

SET DATABASE GC

set database gc statement

```
<set database gc statement> ::= SET DATABASE GC <unsigned integer literal>
```

An optional property which forces calls to `System.gc()` after the specified number of row operations. The default value for this property is 0, which means no System.gc() calls. Usual values for this property range from 10000 depending on the system and the memory allocation. This property may be useful in some in-process deployments, especially with older JVM implementations.

Only a user with the DBA role can execute this statement.

SET DATABASE SQL SIZE

set database sql size statement

```
<set database sql size statement> ::= SET DATABASE SQL SIZE { TRUE | FALSE }
```

Enable or disable enforcement of column sizes for CHAR and VARCHAR columns. The default is TRUE, meaning table definition must contain `VARCHAR(n)` instead of `VARCHAR`.

Only a user with the DBA role can execute this statement.

SET DATABASE SQL NAMES

set database sql names statement

```
<set database sql names statement> ::= SET DATABASE SQL NAMES { TRUE | FALSE }
```

Enable or disable full enforcement of the rule that prevents SQL keywords being used for database object names such as columns and tables. The default is FALSE, meaning disable.

Only a user with the DBA role can execute this statement.

SET DATABASE SQL REFERENCES

set database sql references statement

```
<set database sql references statement> ::= SET DATABASE SQL REFERENCES { TRUE
| FALSE }
```

This command can enable or disable full enforcement of the rule that prevents ambiguous column references in SQL statements (usually SELECT statements). A column reference is ambiguous when it is not qualified by a table name or table alias and can refer to more than one column in a JOIN list.

The property is FALSE by default. It is better to enable this check while development, to improve the quality and correctness of SQL statements.

Only a user with the DBA role can execute this statement.

SET DATABASE REFERENTIAL INTEGRITY

set database referential integrity statement

```
<set database referential integrity statement> ::= SET DATABASE REFERENTIAL
INTEGRITY { TRUE | FALSE }
```

This command enables or disables the enforcement of referential integrity constraints (foreign key constraints), check constraints apart from NOT NULL and triggers. By default, referential integrity constraints are checked.

The only legitimate use of this statement is before importing large amounts of external data into tables that have existing FOREIGN KEY constraints. After import, the statement must be used again to enable constraint enforcement.

If you are not sure the data conforms to the constraints, run queries to verify all rows conform to the FOREIGN KEY constraints and take appropriate actions for the rows that do not conform.

A query example to return the rows in a foreign key table that have no parent is given below:

Example 11.5. Finding foreign key rows with no parents after a bulk import

```
SELECT * FROM foreign_key_table LEFT OUTER JOIN primary_key_table
   ON foreign_key_table.fk_col = primary_key_table.pk_col WHERE primary_key_table.pk_col IS NULL
```

Only a user with the DBA role can execute this statement.

SET DATABASE UNIQUE NAME

set database unique name

```
<set database unique name statement> ::= SET DATABASE UNIQUE NAME <identifier>
```

Each HyperSQL catalog (database) has an engine-generated internal name. This name is based on the time of creation of the database and is exactly 16 characters. The name is used for in log events sent to external logging frameworks. This name can be changed by an administrator. The new name must be exactly 16 characters long.

SET DATABASE TRANSACTION CONTROL

set database transaction control statement

```
<set database transaction control statement> ::= SET DATABASE TRANSACTION CONTROL
{ LOCKS | MVLOCKS | MVCC }
```

Set the concurrency control system for the database. It can be issued only when all sessions have been committed or rolled back. This command and its modes is discussed in the Sessions and Transactions chapter.

Only a user with the DBA role can execute this statement.

SET FILES BACKUP INCREMENT

set files backup increment statement

```
<set database backup increment statement> ::= SET FILES BACKUP INCREMENT { TRUE
| FALSE }
```

Older versions of HSQLDB perform a backup of the .data file before its contents are modified and the whole .data file is saved in a compressed form when a CHECKPOINT or SHUTDOWN is performed. This takes a long time when the size of the database exceeds 100 MB or so (on an average 2010 computer, you can expect a backup speed of 20MB / s or more).

The alternative is backup in increments, just before some part of the .data file is modified. In this mode, no backup is performed at CHECKPIONT or SHUTDOWN. This mode is preferred for large databases which are opened and closed frequently.

The default mode is TRUE. If the old method of backup is preferred, the mode can be set FALSE.

Only a user with the DBA role can execute this statement.

SET FILES CACHE ROWS

set files cache rows statement

```
<set files cache rows statement> ::= SET FILES CACHE ROWS <unsigned integer
literal>
```

Sets the maximum number of rows (of CACHED tables) held in the memory cache.

Only a user with the DBA role can execute this statement.

SET FILES CACHE SIZE

set files cache size statement

```
<set files cache size statement> ::= SET FILES CACHE SIZE <unsigned integer
literal>
```

Sets maximum amount of data (of CACHED tables) in kilobytes held in the memory cache.

Only a user with the DBA role can execute this statement.

SET FILES DEFRAG

set files defrag statement

```
<set files defrag statement> ::= SET FILES DEFRAG <unsigned integer literal>
```

Sets the threshold for performing a DEFRAG during a checkpoint. The `<unsigned integer literal>` is the percentage of abandoned space in the `*.data` file. When a CHECKPOINT is performed either as a result of the `.log` file reaching the limit set by `SET FILES LOG SIZE m`, or by the user issuing a CHECKPOINT command, the amount of space abandoned since the database was opened is checked and if it is larger than specified percentage, a CHECKPOINT DEFRAG is performed instead of a CHECKPOINT.

The default is 0, which indicates no DEFRAG. Useful values are between 10 to 50

Only a user with the DBA role can execute this statement.

SET FILES LOG

set files log statement

```
<set files log statement> ::= SET FILES LOG { TRUE | FALSE }
```

Sets logging of database operations on or off. Turning logging off is for special usage, such as temporary cache usage.

Only a user with the DBA role can execute this statement.

SET FILES LOG SIZE

set files log size statement

```
<set files log size statement> ::= SET FILES LOG SIZE <unsigned integer literal>
```

Sets the maximum size in MB of the `*.log` file to the specified value. The default maximum size is 50 MB. If the value is zero, no limit is used for the size of the file. When the size of the file reaches this value, a CHECKPOINT is performed and the the `*.log` file is cleared to size 0.

Only a user with the DBA role can execute this statement.

SET FILES BACKUP INCREMENT

set files backup increment statement

```
<set files increment backup statement> ::= SET FILES INCREMENT BACKUP { TRUE
| FALSE }
```

This specifies the method for internal backup operation. The default is true.

During updates, the contents of the .data file is modified. When this property is true, the modified contents are backed up gradually. This causes a marginal slowdown in operations, but allows fast checkpoint and shutdown with large .data files.

When the property is false, the .data file is backed up entirely at the time of checkpoint and shutdown. Up to version 1.8.0, HSQLDB supported only full backup. Version 1.8.1 supports incremental backup.

Only a user with the DBA role can execute this statement.

SET FILES NIO

set files nio

```
<set files nio statement> ::= SET FILES NIO { TRUE | FALSE }
```

Changes the access method of the .data file. The default is TRUE and uses the Java nio classes to access the file.

Only a user with the DBA role can execute this statement.

SET FILES WRITE DELAY

set files write delay statement

```
<set files write delay statement> ::= SET FILES WRITE DELAY {{ TRUE | FALSE }
| <seconds value> | <milliseconds value> MILLIS}
```

Set the WRITE DELAY property of the database. The WRITE DELAY controls the frequency of file sync for the log file. When WRITE_DELAY is set to FALSE or 0, the sync takes place immediately at each COMMIT. WRITE DELAY TRUE performs the sync once every 10 seconds (which is the default). A numeric value can be specified instead.

The purpose of this command is to control the amount of data loss in case of a total system crash. A delay of 1 second means at most the data written to disk during the last second before the crash is lost. All data written prior to this has been synced and should be recoverable.

A write delay of 0 impacts performance in high load situations, as the engine has to wait for the file system to catch up.

To avoid this, you can set write delay down to 10 milliseconds.

Each time the SET FILES WRITE DELAY statement is executed with any value, a sync is immediately performed. Only a user with the DBA role can execute this statement.

Only a user with the DBA role can execute this statement.

SET FILES SCALE

set files scale

```
<set files scale statement> ::= SET FILES SCALE <scale value>
```

Changes the scale factor for the .data file. The default scale is 8 and allows 16GB of data storage capacity. The scale can be increased in order to increase the maximum data storage capacity. The scale values 8, 16, 32, 64 and 128 are allowed. Scale value 128 allows a maximum capacity of 256GB.

This command can be used only when there is no data in CACHED tables.

Only a user with the DBA role can execute this statement.

SET FILES LOB SCALE

set files lob scale

```
<set files lob scale statement> ::= SET FILES LOB SCALE <scale value>
```

Changes the scale factor for the .lobs file. The scale is interpreted in kilobytes. The default scale is 32 and allows 64TB of lob data storage capacity. The scale can be reduced in order to improve storage efficiency. If the lobs are a lot smaller than 32 kilobytes, reducing the scale will reduce wasted space. The scale values 1, 2, 4, 8, 16, 32 are allowed. For example if the average size of lobs is 4 kilobytes, the default scale of 32 will result in 28KB wasted space for each lob. Reducing the lob scale to 2 will result in average 1KB wasted space for each lob.

This command can be used only when there is no lob in the database.

Only a user with the DBA role can execute this statement.

Chapter 12. Properties

Fred Toussi, The HSQL Development Group

$Revision: 3626 $

Connections

The normal method of accessing a HyperSQL catalog is via the JDBC Connection interface. An introduction to different methods of providing database services and accessing them can be found in the SQL Language chapter. Details and examples of how to connect via JDBC are provided in our JavaDoc for `JDBCConnection`.

A uniform method is used to distinguish between different types of connection. The common driver identifier is `jdbc:hsqldb:` followed by a protocol identifier (`mem: file: res: hsql: http: hsqls: https:`) then followed by host and port identifiers in the case of servers, then followed by database identifier. Additional property / value pairs can be appended to the end of the URL, separated with semicolons.

Table 12.1. HyperSQL URL Components

Driver and Protocol	Host and Port	Database
`jdbc:hsqldb:mem:`	not available	`accounts`
Lowercase, single-word identifier creates the in-memory database when the first connection is made. Subsequent use of the same Connection URL connects to the existing DB. The old form for the URL, `jdbc:hsqldb:.` creates or connects to the same database as the new form for the URL, `jdbc:hsqldb:mem:.`		

`jdbc:hsqldb:file:`	not available	`mydb`
		 `/opt/db/accounts`
		 `C:/data/mydb`

The file path specifies the database file. In the above examples the first one refers to a set of mydb.* files in the directory where the `java` command for running the application was issued. The second and third examples refer to absolute paths on the host machine.		
`jdbc:hsqldb:res:`	not available	`/adirectory/dbname`
Database files can be loaded from one of the jars specified as part of the `Java` command the same way as resource files are accessed in Java programs. The `/adirectory` above stands for a directory in one of the jars.		

`jdbc:hsqldb:hsql:`
 `jdbc:hsqldb:hsqls:`
 `jdbc:hsqldb:http:`
 `jdbc:hsqldb:https:`
	`//localhost`
	 `//192.0.0.10:9500`
	 `//dbserver.somedomain.com`
		`/an_alias`
		 `/enrollments`
		 `/quickdb`

The host and port specify the IP address or host name of the server and an optional port number. The database to connect to is specified by an alias. This alias is a lowercase string defined in the `server.properties` file to		

Driver and Protocol	Host and Port	Database
refer to an actual database on the file system of the server or a transient, in-memory database on the server. The following example lines in `server.properties` or `webserver.properties` define the database aliases listed above and accessible to clients to refer to different file and in-memory databases. The old form for the server URL, e.g., `jdbc:hsqldb:hsql//localhost` connects to the same database as the new form for the URL, `jdbc:hsqldb:hsql//localhost/` where the alias is a zero length string.		

Connection properties

Each JDBC Connection to a database can specify connection properties. The properties user and password are always required. In 2.0 the following optional properties can also be used.

Connection properties are specified either by establishing the connection via the method call below, or the property can be appended to the full Connection URL.

```
    DriverManager.getConnection (String url, Properties info);
```

Table 12.2. Connection Properties

get_column_name	`true`	column name in ResultSet
This property is used for compatibility with other JDBC driver implementations. When true (the default), `ResultSet.getColumnName(int c)` returns the underlying column name. This property can be specified differently for different connections to the same database. The default is true. When the property is false, the above method returns the same value as `ResultSet.getColumnLabel(int column)` Example below: `    jdbc:hsqldb:hsql://localhost/enrollments;get_column_name=false` When a ResultSet is used inside a user-defined stored procedure, the default, true, is always used for this property.		
ifexists	`false`	connect only if database already exists
Has an effect only with *mem:* and *file:* database. When true, will not create a new database if one does not already exist for the URL. When the property is false (the default), a new *mem:* or *file:* database will be created if it does not exist. Setting the property to true is useful when troubleshooting as no database is created if the URL is malformed. Example below: `    jdbc:hsqldb:file:enrollments;ifexists=true`		
shutdown	`false`	shut down the database when the last connection is closed
If this property is `true`, when the last connection to a database is closed, the database is automatically shut down. The property takes effect only when the first connection is made to the database. This means the connection that opens the database. It has no effect if used with subsequent connections.		

This command has two uses. One is for test suites, where connections to the database are made from one JVM context, immediately followed by another context. The other use is for applications where it is not easy to configure the environment to shutdown the database. Examples reported by users include web application servers, where the closing of the last connection coincides with the web app being shut down.

```
jdbc:hsqldb:file:enrollments;shutdown=true
```

In addition, when a connection to an *in-process* database creates a new database, or opens an existing database (i.e. it is the first connection made to the database by the application), all the user-defined database properties can be specified as URL properties. This can be used to specify properties to enforce more strict SQL adherence, or to change cache_scale or similar properties before the database files are created. However, for new databases, it is recommended to use the SET PROPERTY command for such settings.

Database Properties in Connection URL and Properties

The database engine has several properties that are listed in the System Management and Deployment Issues chapter. These properties can be changed via SQL commands after a connection is made to the database. It is possible to specify all of these properties in the connection properties on as part of the URL string when the first connection is made to a new file: or mem: database. This allows the properties to be set without using any SQL commands. The corresponding SQL command is given for each property.

Management of properties has changed since version 1.8. The old SET PROPERTY does not change a property and is retained to simplify application upgrades.

In the example URL below, two properties are set for the first connection to a new database. If the properties are used for connection to an existing database, they are ignored.

```
jdbc:hsqldb:file:enrollments;hsqldb.cache_rows=10000;hsqldb.nio_data_file=false
```

In the table below, database properties that can be used as part of the URL below are given. For each property that can also be set with an SQL statement, the statement is also given. These statements are described in the System Management and Deployment Issues chapter.

Table 12.3. Database-specific Property File Properties

Value	Default	Description
check_props	`false`	checks the validity of the connection properties
If the property is true, every database property that is specified on the URL or in connection properties is checked and if it is not used correctly, an error is returned		
`this property cannot be set with an SQL statement`		
crypt_lobs	`false`	encryption of lobs
If the property is true, the contents of the .lobs file is encrypted as well.		
`this property cannot be set with an SQL statement`		
crypt_key	`none`	encryption
The cipher key for an encrypted database		

Value	Default	Description
this property cannot be set with an SQL statement		
crypt_provider	none	encryption
The fully-qualified class name of the cryptography provider. This property is not used for the default security provider.		
this property cannot be set with an SQL statement		
crypt_type	none	encryption
The cipher specification.		
this property cannot be set with an SQL statement		
read_only	false	readonly database
This property is a special property that can be added manually to the .properties file, or included in the URL or connection properties. When this property is true, the database becomes readonly.		
this property cannot be set with an SQL statement		
files_read_only	false	readonly files database
This property is used similarly to the hsqldb.read_only property. When this property is true, CACHED and TEXT tables are readonly but memory files are not. Any change to the data is not persisted to database files.		
this property cannot be set with an SQL statement		
hsqldb.log_data	true	recovery log
This property can be set to false when database recovery in the event of an unexpected crash is not necessary. A database that is used as a temporary cache is an example. Regardless of the value of this property, if there is a proper shutdown of the database, all the change data is stored.		
this property cannot be set with an SQL statement		
sql.enforce_names	false	enforcing SQL keywords
This property, when set true, prevents SQL keywords being used for database object names such as columns and tables.		
SET DATABASE SQL NAMES { TRUE | FALSE }		
sql.enforce_size	true	trimming and padding string columns.
This property is the same as sql.enforce_strict_size		
sql.enforce_strict_size	true	size enforcement and padding string columns
Conforms to SQL standards for size and precision of data types. When true, all CHARACTER, VARCHAR, NUMERIC and DECIMAL values that are in a row affected by an INSERT INTO or UPDATE statement are checked		

Value	Default	Description
against the size specified in the SQL table definition. An exception is thrown if the value is too long. Also all CHARACTER values that are shorter than the specified size are padded with spaces. `SET DATABASE SQL SIZE { TRUE	FALSE }`		
sql.enforce_refs	`false`	enforcing column reference disambiguation
This property, when set true, causes an error when an SQL statements contains column references that can be resovled by more than one table name or alias. In effect forces such column references to have a table name or table alias qualifier. `SET DATABASE SQL REFERENCES { TRUE	FALSE }`		
runtime.gc_interval	0	forced garbage collection
This setting forces garbage collection each time a set number of result set row or cache row objects are created. The default, "0" means no garbage collection is forced by the program. `SET DATABASE GC <numeric value>`		
hsqldb.default_table_type	`memory`	type of table created with unqualified CREATE TABLE
The CREATE TABLE command results in a MEMORY table by default. Setting the value *cached* for this property will result in a cached table by default. The qualified forms such as CREATE MEMORY TABLE or CREATE CACHED TABLE are not affected at all by this property. `SET DATABASE DEFAULT TABLE TYPE { CACHED	MEMORY }`		
hsqldb.applog	0	application logging level
The default level 0 indicates no logging. Level 1 results in events related to persistence to be logged, including any failures. Level 2 indicates all events, including ordinary events. The events are logged in a file ending with ".app.log". `SET DATABASE EVENT LOG LEVEL { 0	1	2 }`		
hsqldb.result_max_memory_rows	0	amount of result rows that are kept in memory
Sets the maximum number of rows of each result set and other internal temporary table that is held in memory. `SET DATABASE DEFAULT RESULT MEMORY ROWS <unsigned integer literal>`		
hsqldb.tx	`locks`	database transaction control mode
Indicates the transaction control mode for the database. The values, locks, mvlocks and mvcc are allowed. `SET DATABASE TRANSACTION CONTROL { LOCKS	MVLOCKS	MVCC }`		

Value	Default	Description
hsqldb.cache_rows	`50000`	maximum number of rows in memory cache

Indicates the maximum number of rows of cached tables that are held in memory.

The value can range between 100-1,000,000. If the value is set via SET FILES then it becomes effective after the next database SHUTDOWN or CHECKPOINT.

The property is changed via the `SET FILES CACHE ROWS nnn` SQL command.

```
SET FILES CACHE ROWS <numeric value>
```

Value	Default	Description
hsqldb.cache_size	`10000`	memory cache size

Indicates the total size (in kilobytes) of rows in the memory cache used with cached tables. This size is calculated as the binary size of the rows, for example an INTEGER is 4 bytes. The actual memory size used by the objects is 2 to 4 times this value. This depends on the types of objects in database rows, for example with binary objects the factor is less than 2, with character strings, the factor is just over 2 and with date and timestamp objects the factor is over 3.

The value can range between 100-1,000,000. The default is 10,000, representing 10,000 kilobytes. If the value is set via SET FILES then it becomes effective after the next database SHUTDOWN or CHECKPOINT.

```
SET FILES CACHE SIZE <numeric value>
```

Value	Default	Description
hsqldb.inc_backup	`true`	incremental backup of data file

During updates, the contents of the .data file are modified. When this property is true, the modified contents are backed up gradually. This causes a marginal slowdown in operations, but allows fast checkpoint and shutdown.

When the property is false, the .data file is backed up entirely at the time of checkpoint and shutdown. Up to version 1.8, HSQLDB supported only full backup.

```
SET FILES INCREMENT BACKUP { TRUE | FALSE }
```

Value	Default	Description
hsqldb.lock_file	`true`	use of lock file

By default, a lock file is created for each file database that is opened for read and write. This property can be specified with the value false to prevent the lock file from being created. This usage is not recommended but may be desirable when flash type storage is used.

```
this property cannot be set with an SQL statement
```

Value	Default	Description
hsqldb.log_data	`true`	logging data change

When `false` is specified, no data logging takes place. A checkpoint or shutdown still rewrites the `.script` file and saves the `.backup` file according to the other settings.

```
SET FILES LOG  { TRUE | FALSE }
```

Value	Default	Description
hsqldb.log_size	50	size of log when checkpoint is performed
The value is the size (in megabytes) that the `.log` file can reach before an automatic checkpoint occurs. A checkpoint rewrites the `.script` file and clears the `.log` file. `SET FILES LOG SIZE <numeric value>`		
hsqldb.nio_data_file	true	use of nio access methods for the .data file
When HyperSQL is compiled and run in Java 1.4 or higher, setting this property to `false` will avoid the use of nio access methods, resulting in somewhat reduced speed. If the data file is larger than 256MB when it is first opened, nio access methods are not used. Also, if the file gets larger than the amount of available computer memory that needs to be allocated for nio access, non-nio access methods are used. If used before defining any CACHED table, it applies immediately, otherwise it comes into effect after a SHUTDOWN and restart or CHECKPOINT. `SET FILES NIO { TRUE	FALSE }`		
hsqldb.translate_dti_types	true	usage of type codes for advanced type datetime types
If the property is true, the datetime WITH TIME ZONE types and INTERVAL types are represented as JDBC datetime types without time zone and the VARCHAR type respectively. `this property cannot be set with an SQL statement`		
hsqldb.write_delay	true	write delay for writing log file entries
If the property is true, the default WRITE DELAY property of the database is used, which is 1000 milliseconds. If the property is false, the WRITE DELAY is set to 0 seconds. The SQL command for this property allows more precise control over the property. `SET FILES WRITE DELAY {{ TRUE	FALSE }	<seconds value>	<milliseconds value> MILLIS`		
hsqldb.write_delay_millis	1000	write delay for writing log file entries
If the property is used, the WRITE DELAY property of the database is set the given value. The SQL command for this property allows the same level of control over the property. `SET FILES WRITE DELAY {{ TRUE	FALSE }	<seconds value>	<milliseconds value> MILLIS`		
textdb.*	0	default properties for new text tables
Properties that override the database engine defaults for newly created text tables. Settings in the text table `SET <tablename> SOURCE <source string>` command override both the engine defaults and the database properties defaults. Individual textdb.* properties are listed in the Text Tables chapter.		

When connecting to an *in-process* database creates a new database, or opens an existing database (i.e. it is the first connection made to the database by the application), all the user-defined database properties listed in this section can be specified as URL properties.

When HSQLDB is used in OpenOffice.org, some property values will have a different default. The properties and values are:

hsqldb.default_table_type=cached   hsqldb.cache_rows=25000;   hsqldb.cache_size=6000;   hsqldb.log_size=10; hsqldb.nio_data_file=false; sql.enforce_strict_size=true

Chapter 13. HyperSQL Network Listeners

Server, WebServer, and Servlet

Fred Toussi, The HSQL Development Group

$Revision: 3601 $

Copyright 2002-2009 Fred Toussi. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQL Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.
Published $Date: 2010-05-31 20:17:47 -0400 (Mon, 31 May 2010) $

Listeners

As described in the Running and Using HyperSQL chapter, network listeners or servers provide connectivity to catalogs from different JVM processes. The HyperSQL listeners support both ipv4 and ipv6 network addressing.

HyperSQL Server

This is the preferred way of running a database server and the fastest one. This mode uses the proprietary *hsql:* communications protocol. The following example of the command for starting the server starts the server with one (default) database with files named "mydb.*" and the public name (alias) of "xdb".

```
java -cp ../lib/hsqldb.jar org.hsqldb.server.Server --database.0 file:mydb --dbname.0 xdb
```

Alternatively, a server.properties file can be used for passing the arguments to the server. This file must be located in the directory where the command is issued.

```
java -cp ../lib/hsqldb.jar org.hsqldb.server.Server
```

The contents of the server.properties file is described in the next section.

HyperSQL HTTP Server

This method of access is used when the computer hosting the database server is restricted to the HTTP protocol. The only reason for using this method of access is restrictions imposed by firewalls on the client or server machines and it should not be used where there are no such restrictions. The HyperSQL HTTP Server is a special web server that allows JDBC clients to connect via HTTP. The server can also act as a small general-purpose web server for static pages.

To run an HTTP server, replace the main class for the server in the example command line above with the following:

```
java -cp ../lib/hsqldb.jar org.hsqldb.server.Server
```

The contents of the server.properties file is described in the next section.

HyperSQL HTTP Servlet

This method of access also uses the HTTP protocol. It is used when a separate servlet engine (or application server) such as Tomcat or Resin provides access to the database. The Servlet Mode cannot be started independently from the servlet engine. The `Servlet` class, in the HSQLDB jar, should be installed on the application server to provide the

connection. The database is specified using an application server property. Refer to the source file `src/org/hsqldb/server/Servlet.java` to see the details.

Both HTTP Server and Servlet modes can only be accessed using the JDBC driver at the client end. They do not provide a web front end to the database. The Servlet mode can serve only a single database.

Please note that you do not normally use this mode if you are using the database engine in an application server. In this situation, connections to a catalog are usually made *in-process*, or using an external HSQL Server instance.

Server and Web Server Properties

Properties files for running the servers are not created automatically. You should create your own files that contain server.property=value pairs for each property. The `server.properties` or `webserver.properties` files must be located in the directory where the command to run the `org.hsqldb.server.Server` class is issued.

In all properties files, values are case-sensitive. All values apart from names of files or pages are required in lowercase (e.g. server.silent=FALSE will have no effect, but server.silent=`false` will work). Supported properties and their default values (if any) are as follows:

Table 13.1. common server and webserver properties

Value	Default	Description
server.database.0	`file:test`	the catalog type, path and file name of the first database file to use
server.dbname.0	`""`	lowercase server alias for the first database file
server.database.n	`NO DEFAULT`	the catalog type, path and file name of the n'th database file in use
server.dbname.n	`NO DEFAULT`	lowercase server alias for the n'th database file
server.silent	`true`	no extensive messages displayed on console
server.trace	`false`	JDBC trace messages displayed on console
server.address	`NO DEFAULT`	IP address of server
server.tls	`false`	Whether to encrypt network stream. If this is set to `true`, then in normal situations you will also need to set properties `system.javax.net.ssl.keyStore` and `system.javax.net.ssl.keyStorePassword`, as documented elsewhere. The value of `server.tls` impacts the default value of `server.port`.
server.remote_open	`false`	Allows opening a database path remotely when the first connection is made

In HyperSQL version 2.0, each server can serve an unlimited number of databases simultaneously. The server.database.0 property defines the filename / path whereas the server.dbname.0 defines the lowercase alias used by clients to connect to that database. The digit 0 is incremented for the second database and so on. Values for the server.database.n property can use the *mem:*, *file:* or *res:* prefixes and connection properties as discussed under CONNECTIONS. For example,

```
database.0=mem:temp;sql.enforce_strict_size=true;
```

Properties or default values specific to `server.properties` are:

Table 13.2. server properties

Value	Default	Description
server.port	`9001 (normal) or 554 (if TLS en-crypted)`	TCP/IP port used for talking to clients. All databases are served on the same port.
server.no_system_exit	`true`	no `System.exit()` call when the database is closed

Properties or default values specific to `webserver.properties` are:

Table 13.3. webserver properties

Value	Default	Description
server.port	`80 (normal) or 443 (if TLS encrypted)`	TCP/IP port used for talking to clients
server.default_page	`index.html`	the default web page for server
server.root	`./`	the location of served pages
.<extension>	`NO DEFAULT`	multiple entries such as `.html=text/html` define the mime types of the static files served by the web server. See the source for `src/org/hsqldb/server/WebServer.java` for a list.

An example of the contents of a `server.properties` file is given below:

```
server.database.0=file:/opt/db/accounts
server.dbname.0=accounts

server.database.1=file:/opt/db/mydb
server.dbname.1=enrollments

server.database.2=mem:adatabase
server.dbname.2=quickdb
```

In the above example, the `server.properties` file indicates that the server provides access to 3 different databases. Two of the databases are file-based, while the third is all-in-memory. The aliases for the databases that the users connect to are `accounts`, `enrollments` and `quickdb`.

All the above properties and their values can be specified on the command line to start the server by omitting the `server.` prefix. If a property/value pair is specified on the command line, it overrides the property value specified in the `server.properties` or `webserver.properties` file.

> **Note**
>
> Upgrading: If you have existing custom properties files, change the values to the new naming convention. Note the use of digits at the end of server.database.n and server.dbname.n properties.

Starting a Server from your application

If you want to start the server from within your application, as opposed to the command line or batch files, you should create an instance of Server or Web Server, then assign the properties and start the Server. An working example of this can be found in the `org.hsqldb.test.TestBase` source. The example below sets the same properties as in the server.properties file example.

```
HsqlProperties p = new HsqlProperties();
```

```
    p.setProperty("server.database.0","file:/opt/db/accounts");
    p.setProperty("server.dbname.0","an_alias");
    // set up the rest of properties
    Server server = new Server();
    server.setProperties(p);
    server.setLogWriter(null); // can use custom writer
    server.setErrWriter(null); // can use custom writer
    server.start();
```

The Server object has several alternative methods for setting databases and their public names. The server should be shutdown using the shutdown() method.

Allowing a Connection to Open a Database

If the `server.remote_open` property is true, the Server works differently from the normal mode. In this mode, it is not necessary to have any databases listed as server.database.0 etc. in the Server startup properties. If there are databases listed, they are opened as normal. The server does not shutdown when the last database is closed.

In this mode, a connection can be established to a database that is not open or does not exist. The server will open the database or create it, then return a connection to the database.

The connection URL must include the path to the database, separated with a semicolon from the alias. In the example below, the database path specified as `file:C:/files/mydatabase` is opened and the database alias `xdb` is assigned to the database. After this, the next connection to the specified alias will connect to the same database.

```
Connection c = DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/xdb;file:C:/files/
mydatabase", "SA", "");
```

The path can be a file: or mem: database.

TLS Encryption

Listener TLS Support (a. k. a. SSL)
Blaine Simpson, The HSQL Development Group
$Revision: 3601 $
Published $Date: 2010-05-31 20:17:47 -0400 (Mon, 31 May 2010) $

This section explains how to encrypt the stream between JDBC network clients and HyperSQL Listeners. If you are running an *in-process* (non-Listener) setup, this chapter does not apply to you.

Requirements

Hsqldb TLS Support Requirements

- Sun Java 2.x and up. (This is probably possible with IBM's Java, but I don't think anybody has attempted to run HSQLDB with TLS under IBM's Java, and I'm sure that nobody in the HSQLDB Development Group has documented how to set up the environment).

- If Java 2.x or 3.x, then you will need to install JSSE . Your server and/or client will start up much slower than that of Java 4.x users. Client-side users will not be able to use the https: JDBC protocol (because the https protocol handler is not implemented in 2.x/3.x Java JSSE; if there is demand, we could work around this).

- A JKS keystore containing a private key , in order to run a Listener.

- If you are running the listener side, then you'll need to run a HSQLDB Server or WebServer Listener instance. It doesn't matter if the underlying database catalogs are new, and it doesn't matter if you are making a new Listener configuration or encrypting an existing Listener configuration. (You can turn encryption on and off at will).

- You need a HSQLDB jar file that was built with JSSE present. If you obtained your HSQLDB 1.7.2-or-later distribution from us, you are all set, because we build with Java 1.4 or later (which contains JSSE). If you build your own jar file with Java 1.3, make sure to install JSSE first.

Encrypting your JDBC connection

At this time, only 1-way, server-cert encryption is tested.

Client-Side

Just use one of the following protocol prefixes.

Hsqldb TLS URL Prefixes

- `jdbc:hsqldb:hsqls://`

- `jdbc:hsqldb:https://`

At this time, the latter will only work for clients running with Java 1.4 or later.

If the listener you wish to connect to is using a certificate approved by your default trust keystore, then there is nothing else to do. If not, then you need to tell Java to "trust" the server cert. (It's a slight over-simplification to say that if the server certificate was purchased, then you are all set; if somebody "signed their own" certificate by self-signing or using a private ca certificate, then you need to set up trust).

First, you need to obtain the cert (only the "public" part of it). Since this cert is passed to all clients, you could obtain it by writing a Java client that dumps it to file, or perhaps by using *openssl s_client*. Since in most cases, if you want to trust a non-commercial cert, you probably have access to the server keystore, I'll show an example of how to get what you need from the server-side JKS keystore.

You may already have an X509 cert for your server. If you have a server keystore, then you can generate a X509 cert like this.

Example 13.1. Exporting certificate from the server's keystore

```
keytool -export -keystore server.store -alias existing_alias -file server.cer
```

In this example, `server.cer` is the X509 certificate that you need for the next step.

Now, you need to add this cert to one of the system trust keystores or to a keystore of your own. See   the Customizing Stores section in JSSERefGuide.html   [http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#CustomizingStores] to see where your system trust keystores are. You can put private keystores anywhere you want to. The following command will add the cert to an existing keystore, or create a new keystore if `client.store` doesn't exist.

Example 13.2. Adding a certificate to the client keystore

```
keytool -import -trustcacerts -keystore trust.store -alias new_alias -file server.cer
```

If you are making a new keystore, you probably want to start with a copy of your system default keystore which you can find somewhere under your `JAVA_HOME` directory (typically `jre/lib/security/cacerts` for a JDK, but I forget exactly where it is for a JRE).

Unless your OS can't stop other people from writing to your files, you probably do not want to set a password on the trust keystore.

If you added the cert to a system trust store, then you are finished. Otherwise you will need to specify your custom trust keystore to your client program. The generic way to set the trust keystore is to set the system property `javax.net.ssl.trustStore` every time that you run your client program. For example

Example 13.3. Specifying your own trust store to a JDBC client

```
java -Djavax.net.ssl.trustStore=/home/blaine/trust.store -jar /path/to/hsqldb.jar dest-urlid
```

This example runs the program SqlTool . SqlTool has built-in TLS support however, so, for SqlTool you can set `truststore` on a per-urlid basis in the SqlTool configuration file.

Note: The hostname in your database URL must match the *Common Name* of the server's certificate exactly. That means that if a site certificate is `admc.com`, you can not use `jdbc:hsqldb:hsqls://localhost` or `jdbc:hsqldb:hsqls://www.admc.com:1100` to connect to it.

If you want more details on anything, see JSSERefGuide.html on Sun's site [http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html], or in the subdirectory `docs/guide/security/jsse` of your Java SE docs.

Server-Side, aka Listener-Side

Get yourself a   JKS keystore containing a private key . Then set properties `server.tls`, `system.javax.net.ssl.keyStore` and `system.javax.net.ssl.keyStorePassword` in your `server.properties` or `webserver.properties` file. Set `server.tls` to `true`, `system.javax.net.ssl.keyStore` to the path of the private key JKS keystore, and `system.javax.net.ssl.keyStorePassword` to the password (of both the keystore and the private key record-- they must be the same). If you specify relative file path values, they will be resolved relative to the `${user.dir}` when the JRE is started.

> ### Caution
>
> If you set any password in a .properties (or any other) file, you need to restrict access to the file. On a good operating system, you can do this like so:
>
> ```
> chmod 600 path/to/server.properties
> ```

The values and behavior of the `system.*` settings above match the usage documented for `javax.net.ssl.keyStorePassword` and `javax.net.ssl.keyStore` in the JSSE docs.

> ### Note
>
> Before version 2.0, HyperSQL depended on directly setting the corresponding JSSE properties. The new idiom is more secure and easier to manage. If you have an old password in a UNIX init script config file, you should remove it.

JSSE

If you are running Java 4.x or later, then you are all set. Java 1.x users, you are on your own (Sun does not provide a JSSE that will work with 1.x). Java 2.x and 3.x users continue...

Go to http://java.sun.com/products/jsse/index-103.html If you agree to the terms and meet the requirements, download the domestic or global JSSE software. All you need from the software distro is the three jar files. If you have a JDK

installation, then move the 3 jar files into the directory `$JAVA_HOME/jre/lib/ext`. If you have a JRE installation, then move the 3 jar files into the directory `$JAVA_HOME/lib/ext`.

Pretty painless.

Making a Private-key Keystore

There are two main ways to do this. Either you can use a certificate signed by a certificate authority, or you can make your own. One thing that you need to know in both cases is, the *Common Name* of the cert has to be the exact hostname that JDBC clients will use in their database URL.

CA-Signed Cert

I'm not going to tell you how to get a CA-signed SSL certificate. That is well documented at many other places.

Assuming that you have a standard pem-style private key certificate, here's how you can use openssl [http://www.openssl.org] and the program `DERImport` to get it into a JKS keystore.

Because I have spent a lot of time on this document already, I am just giving you an example.

Example 13.4. Getting a pem-style private key into a JKS keystore

```
openssl pkcs8 -topk8 -outform DER -in Xpvk.pem -inform PEM -out Xpvk.pk8 -nocrypt

openssl x509 -in Xcert.pem -out Xcert.der -outform DER

java DERImport new.keystore NEWALIAS Xpvk.pk8 Xcert.der
```

> **Important**
>
> Make sure to set the password of the key exactly the same as the password for the keystore!

You need the program `DERImport.class` of course. Do some internet searches to find `DERImport.java` or `DERImport.class` and download it.

If DERImport has become difficult to obtain, I can write a program to do the same thing-- just let me know.

Non-CA-Signed Cert

Run `man keytool` or see the Creating a Keystore section of JSSERefGuide.html [http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#CreateKeystore].

Automatic Server or WebServer startup on UNIX

If you are on UNIX and want to automatically start and stop a Server or WebServer running with encryption, set the `system.javax.net.ssl.keyStore` and `system.javax.net.ssl.keyStorePassword` properties as instructed above, and follow the instructions in the HyperSQL on UNIX chapter, paying close attention to the TLS-related comments in the template config file.

If you are using a private server certificate, make sure to also set the trust store filepath for relevant urlids in your RC file, as explained in the sample config file .

Network Access Control

Aka Server ACLs

JDBC connections will always be denied if the supplied user and password are not found in the target catalog. But an HyperSQL listener can also restrict access at the listener level, even protecting private catalogs which have insecure (or default) passwords. If you have an *in-process* setup, this section of the Guide doesn't apply to you.

Many (in fact, most) distributed database applications don't have application clients connect directly to the database, but instead encapsulate access in a controlling process. For example, a web app will usually access the data source on behalf of users, with end-user web browsers never accessing the database directly. In these cases and others, the security benefits of restricting listener access to specific source addresses is well worth the effort. ACLs work by restricting access according to the source address of the incoming connection request. This is efficient because the database engine never even gets the request until it is approved by the ACL filter code.

The sample file `sample/acl.txt` in your HyperSQL distribution explains how to write an ACL file.

```
# $Id: acl.txt 826 2009-01-17 05:04:52Z unsaved $

# Sample HyperSQL Network Listener ACL file.
# Specify "allow" and "deny" rules
# For address specifications, individual addresses, host names, and
# network addresses with /bit suffix are allowed, but read the caveat about
# host names below, under the sample "localhost" rule.

# Blank lines ignored.
    # Lines with # as the first non-whitespace character are ignored.


allow 2001:db8::/32
# Allow this 32-bit ipv4 subnet

allow localhost
# You should use numerical addresses in ACL files, unless you are certain that
# the name will always be known to your network address resolution system
# (assume that you will lose Internet connectivity at some time).
# With a default name resolution setup on UNIX, you are safe to use names
# defined in your /etc/hosts file.

deny 192.168.101.253
# Deny a single IP address.
# In our example, 192.168.101.0/24 is our local, organizational network.
# 192.168.101.253 is the IP address of our Intern's PC.
# The Intern does not have permission to access our databases directly.

allow 192.168.101.0/24

# Any ipv4 or ipv6 candidate address not matched above will be denied
```

You put your file wherever it is convenient for you, and specify that path with the property `server.acl` or `webserver.acl` in your `server.properties` or `webserver.properties` file (depending on whether your listener instance is a `Server` or `WebServer`). You can specify the ACL file path with an absolute or relative path. If you use a relative path, it must be relative to the `.properties` file. It's often convenient to name the ACL file `acl.txt`, in the same directory as your `.properties` file and specify the property value as just `acl.txt`. This file name is intuitive, and things will continue to work as expected if you move or copy the entire directory.

Warning

If your `Server` or `WebServer` was started with a `*.acl` property, changes afterwards to the ACL file will be picked up immediately by your listener instance. You are advised to use the procedure below to prevent partial edits or mistakes from crippling your running server.

When you edit your ACL file, it is both more convenient and more secure to test it as explained here before activating it. You could, of course, test an ACL file by editing it in-place, then trying to connect to your listener with JDBC

clients from various source addresses. Besides being mightily laborious and boring, with this method it is very easy to accidentally open access to all source addresses or to deny access to all users until you fix incorrect ACL entries.

The suggested method of creating or changing ACLs is to work with an inactive file (for new ACL files, just don't enable the `*.acl` property yet; for changing an existing file, just copy it to a temporary file and edit the temporary file). Then use the `ServerAcl` class to test it.

Example 13.5. Validating and Testing an ACL file

```
java -cp path/to/hsqldb.jar org.hsqldb.server.ServerAcl path/to/acl.txt
```

If the specified ACL file fails validation, you will be given details about the problem. Otherwise, the validated rules will be displayed (including the implicit, default deny rules). You then type in host names and addresses, one-per-line. Each name or address is tested as if it were a HyperSQL network client address, using the same exact method that the HyperSQL listener will use. (HyperSQL listeners use this same `ServerAcl` class to test incoming source addresses). `ServerAcl` will report the rule which matches and whether access is denied or allowed to that address.

If you have edited a copy of an existing ACL file (as suggested above), then overwrite your live ACL file with your new, validated ACL file. I.e., copy your temp file over top of your live ACL file.

`ServerAcl` can be run in the same exact way described above, to troubleshoot runtime access issues. If you use an ACL file and a user or application can't get a connection to the database, you can run `ServerAcl` to quickly and definitively find if the client is being prohibited by an ACL rule.

Chapter 14. HyperSQL on UNIX

How to quickly get a HyperSQL (aka HSQLDB) Listener up and running on UNIX, including Mac OS X

Blaine Simpson

$Revision: 3360 $
Published $Date: 2009-12-16 10:03:31 -0500 (Wed, 16 Dec 2009) $

Purpose

This chapter explains how to quickly install, run, and use a HyperSQL Listener (aka Server) on UNIX.

Note that, unlike a traditional database server, there are many use cases where it makes sense to run HyperSQL without any listener. This type of setup is called *in-process*, and is not covered here, since there is no UNIX-specific setup in that case.

I intend to cover what I think is the most common UNIX setup: To run a multi-user, externally-accessible catalog with permanent data persistence. (By the latter I mean that data is stored to disk so that the catalog data will persist across process shutdowns and startups). I also cover how to run the Listener as a system daemon.

When I give sample shell commands below, I use commands which will work in Bourne-compatible shells, including Bash and Korn. Users who insist on using the inferior C-shells will need to convert.

Installation

Go to http://sourceforge.net/projects/hsqldb and click on the "files" link. You want the current version. I can't be more specific because SourceForge/Geeknet are likely to continue changing their interface. See if there's a distribution for the current HSQLDB version in the format that you want.

If you want a binary package and we either don't provide it, or you prefer somebody else's build, you should still find out the current version of HyperSQL available at SourceForge. It's very likely that you can find a binary package for your UNIX variant with your OS distributor, http://www.jpackage.org/ , http://sunfreeware.com/ , etc. Nowadays, most UNIXes have software package management systems which check Internet repositories. Just search the repositories for "hsqldb" and "hypersql". The challenge is to find an *up-to-date* package. You will get better features and support if you work with the current stable release of HyperSQL. (In particular, HyperSQL version 2.0.0 added tons of new features). Pay attention to what JVM versions your binary package supports. Our builds (version 2.0 and later) document the Java version it was built with in the file `doc/index.html`, but you can't depend on this if somebody else assembled your distribution. Java jar files are generally compatible with the same or greater major versions. For example, if your `hsqldb.jar` was built with Java 1.3.6-11, then it is compatible with Java versions 1.3.* and greater.

> ### Note
>
> It could very well happen that some of the file formats which I discuss below are not in fact offered. If so, then we have not gotten around to building them.

Binary installation depends on the package format that you downloaded.

Installing from a .pkg.Z file   This package is only for use by a Solaris super-user. It's a System V package. Download then uncompress the package with uncompress or gunzip

```
    uncompress filename.pkg.Z
```

You can read about the package by running

```
    pkginfo -l -d filename.pkg
```

Run pkgadd as root to install.

```
    pkgadd -d filename.pkg
```

Installing from a BSD Port or Package You're on your own. I find everything much easier when I install software to BSD without their package management systems.

Installing from a .rpm file Just skip this section if you know how to install an RPM. If you found the RPM using a software management system, then just have it install it. The remainder of item explains a generic command-line method which should work with any Linux variant. After you download the rpm, you can read about it by running

```
    rpm -qip /path/to/file.rpm
```

Rpms can be installed or upgraded by running

```
    rpm -Uvh /path/to/file.rpm
```

as root. Suse users may want to keep Yast aware of installed packages by running rpm through Yast: `yast2 -i /path/to/file.rpm`.

Installing from a .zip file Extract the zip file in an ancestor directory of the new HSQLDB home. You don't need to create the HSQLDB_HOME directory because the extraction will create a version-labelled directory, and the subdirectory "hsqldb". This "hsqldb" directory is your HSQLDB_HOME, and you can move it to wherever you wish. If you will be upgrading or maintaining multiple versions of HyperSQL, you will want to retain the version number in the directory tree somehow.

```
    cd ancestor/of/new/hsqldb/home
    unzip /path/to/file.zip
```

All the files in the zip archive will be extracted to underneath a new subdirectory named like `hsqldb-2.0.2a/hsqldb`.

Take a look at the files you installed. (Under `hsqldb` for zip file installations. Otherwise, use the utilities for your packaging system). The most important file of the HyperSQL system is `hsqldb.jar`, which resides in the subdirectory `lib`. Depending on who built your distribution, your file name may have a version label in it, like `hsqldb-1.2.3.4.jar`.

! Important

For the purposes of this chapter, I define HSQLDB_HOME to be the parent directory of the lib directory that contains `hsqldb.jar`. E.g., if your path to `hsqldb.jar` is `/a/b/hsqldb/lib/hsqldb.jar`, then your HSQLDB_HOME is `/a/b/hsqldb`.

Furthermore, unless I state otherwise, all local file paths that I give are relative to the HSQLDB_HOME.

If the description of your distribution says that the `hsqldb.jar` file will work for your Java version, then you are finished with installation. Otherwise you need to build a new `hsqldb.jar` file.

If you followed the instructions above and you still don't know what Java version your `hsqldb.jar` supports, then try reading documentation files like `readme.txt`, `README.TXT`, `INSTALL.txt` etc. (As I said above, our newer distributions always document the Java version for the build, in the file `doc/index.html`). If that still doesn't help, then you can just try your `hsqldb.jar` and see if it works, or build your own.

To use the supplied `hsqldb.jar`, just skip to the   next section of this document . Otherwise build a new `hsqldb.jar`.

Procedure 14.1. Building hsqldb.jar

1. If you don't already have Ant, download the latest stable binary version from http://ant.apache.org . cd to where you want Ant to live, and extract from the archive with

    ```
    unzip /path/to/file.zip
    ```

    or

    ```
    tar -xzf /path/to/file.tar.gz
    ```

    or

    ```
    bunzip2 -c /path/to/file.tar.bz2 | tar -xzf -
    ```

    Everything will be installed into a new subdirectory named `apache-ant-` + `version`. You can rename the directory after the extraction if you wish.

2. Set the environmental variable `JAVA_HOME` to the base directory of your Java JRE or SDK, like

    ```
    export JAVA_HOME; JAVA_HOME=/usr/java/j2sdk1.4.0
    ```

    The location is entirely dependent upon your variety of UNIX. Sun's rpm distributions of Java normally install to `/usr/java/something`. Sun's System V package distributions of Java (including those that come with Solaris) normally install to `/usr/something`, with a sym-link from `/usr/java` to the default version (so for Solaris you will usually set JAVA_HOME to `/usr/java`).

3. Remove the existing file HSQLDB_HOME `/lib/hsqldb.jar`.

4. cd to HSQLDB_HOME `/build`. Make sure that the bin directory under your Ant home is in your search path. Run the following command.

    ```
    ant hsqldb
    ```

    This will build a new HSQLDB_HOME `/lib/hsqldb.jar`.

See the Building HyperSQL Jars appendix if you want to build anything other than `hsqldb.jar` with all default settings.

Setting up a HyperSQL Persistent Database Catalog and a HyperSQL Network Listener

If you installed from an OS-specific package, you may already have a catalog and listener pre-configured. See if your package includes a file named `server.properties` (make use of your packaging utilities). If you do, then I suggest that you still read this section while you poke around, in order to understand your setup.

1. Select a UNIX user to run the database process (JVM) as. If this database is for the use of multiple users, or is a production system (or to emulate a production system), you should dedicate a UNIX user for this purpose. In my

examples, I use the user name `hsqldb`. In this chapter, I refer to this user as the `HSQLDB_OWNER`, since that user will own the database catalog files and the JVM processes.

If the account doesn't exist, then create it. On all system-5 UNIXes and most hybrids (including Linux), you can run (as root) something like

```
useradd -c 'HSQLDB Database Owner' -s /bin/bash -m hsqldb
```

(BSD-variant users can use a similar `pw useradd hsqldb...` command).

2. Become the `HSQLDB_OWNER`. Copy the sample file `sample/server.properties` to the `HSQLDB_OWNER`'s home directory and rename it to `server.properties`. (As a final reminder, "sampleserver.properties" is a relative path, so it is understood to be relative to your `HSQLDB_HOME`).

```
# Hsqldb Server cfg file.
# See the HyperSQL Network Listeners chapter of the HyperSQL User Guide.

# Each server.database.X setting defines a database "catalog".
# I.e., an independent set of data.
# Each server.database.X setting corresponds exactly to the jdbc:hsqldb:*
# JDBC URL you would use if you wanted to get a direct (In-Process)
# Connection to the catalog instead of "serving" it.

server.database.0=file:db0/db0
# I suggest that, for every file: catalog you define, you add the
# connection property "ifexists=true" after the database instance
# is created (which happens simply by starting the Server one time).
# Just append ";ifexists=true" to the file: URL, like so:
# server.database.0=file:db0/db0;ifexists=true

# server.dbname.0 defaults to "" (i.e. server.dbname.n for n==0), but
# the catalog definition n will be entirely ignored for n > 0 if you do not
# set server.dbname.n.  I.e. dbname setting is required for n > 0, though it
# may be set to blank (e.g. "server.dbname.3=")
```

Since the value of the first database (server.database.0) begins with *file:*, the catalog will be persisted to a set of files in the specified directory with names beginning with the specified name. Set the path to whatever you want (relative paths will be relative to the directory containing the properties file). You can read about how to specify other catalogs of various types, and how to make settings for the listen port and many other things in other chapters of this guide.

3. Set and export the environmental variable `CLASSPATH` to the value of `HSQLDB_HOME` (as described above) plus "/lib/hsqldb.jar", like

```
export CLASSPATH; CLASSPATH=/path/to/hsqldb/lib/hsqldb.jar
```

In `HSQLDB_OWNER`'s home directory, run

```
nohup java org.hsqldb.server.Server &
```

This will start the Listener process in the background, and will create your new database catalog "db0". Continue on when you see the message containing `HSQLDB server... is online`. `nohup` just makes sure that the command will not quit when you exit the current shell (omit it if that's what you want to do).

Accessing your Database

We're going to use SqlTool to access the database, so you will need the file `sqltool.jar` in addition to `hsqldb.jar`. If `sqltool.jar` isn't already sitting there beside `hsqldb.jar` (they both come pre-built), build it exactly as you would build `hsqldb.jar`, except use ant target `sqltool`. If your distribution came with a sqltool

jar file with a version label, like `sqltool-1.2.3.4.jar`, that's fine-- use that file whenever I say `sqltool.jar` below.

Copy the file `sample/sqltool.rc` to the HSQLDB_OWNER's home directory. Use `chmod` to make the file readable and writable only to HSQLDB_OWNER.

```
# $Id: sqltool.rc 3353 2009-12-15 19:52:13Z unsaved $

# This is a sample RC configuration file used by SqlTool, DatabaseManager,
# and any other program that uses the org.hsqldb.lib.RCData class.
# See the documentation for SqlTool for various ways to use this file.

# If you have the least concerns about security, then secure access to
# your RC file.

# You can run SqlTool right now by copying this file to your home directory
# and running
#    java -jar /path/to/sqltool.jar mem
# This will access the first urlid definition below in order to use a
# personal Memory-Only database.
# "url" values may, of course, contain JDBC connection properties, delimited
# with semicolons.
# As of revision 3347 of SqlFile, you can also connect to datasources defined
# here from within an SqlTool session/file with the command "\j urlid".

# You can use Java system property values in this file like this:  ${user.home}

# The only feature added recently is the optional "transiso" setting,
# which may be set to an all-caps transaction isolation level as listed
# in the Java API Spec for java.sql.Connection.
# Windows users are advised to use forward slashes instead of reverse slashes,
# and to avoid paths containing spaces or other funny characters.  (This
# recommendation applies to any Java app, not just SqlTool).

# A personal Memory-Only (non-persistent) database.
urlid mem
url jdbc:hsqldb:mem:memdbid
username SA
password

# A personal, local, persistent database.
urlid personal
url jdbc:hsqldb:file:${user.home}/db/personal;shutdown=true
username SA
password
transiso TRANSACTION_READ_COMMITTED
# When connecting directly to a file database like this, you should
# use the shutdown connection property like this to shut down the DB
# properly when you exit the JVM.

# This is for a hsqldb Server running with default settings on your local
# computer (and for which you have not changed the password for "SA").
urlid localhost-sa
url jdbc:hsqldb:hsql://localhost
username SA
password



# Template for a urlid for an Oracle database.
# You will need to put the oracle.jdbc.OracleDriver class into your
# classpath.
# In the great majority of cases, you want to use the file classes12.zip
# (which you can get from the directory $ORACLE_HOME/jdbc/lib of any
# Oracle installation compatible with your server).
# Since you need to add to the classpath, you can't invoke SqlTool with
```

```
# the jar switch, like "java -jar .../hsqldb.jar..." or
# "java -jar .../hsqlsqltool.jar...".
# Put both the HSQLDB jar and classes12.zip in your classpath (and export!)
# and run something like "java org.hsqldb.util.SqlTool...".

#urlid cardiff2
#url jdbc:oracle:thin:@aegir.admc.com:1522:TRAFFIC_SID
#username blaine
#password secretpassword
#driver oracle.jdbc.OracleDriver



# Template for a TLS-encrypted HSQLDB Server.
# Remember that the hostname in hsqls (and https) JDBC URLs must match the
# CN of the server certificate (the port and instance alias that follows
# are not part of the certificate at all).
# You only need to set "truststore" if the server cert is not approved by
# your system default truststore (which a commercial certificate probably
# would be).

#urlid tls
#url jdbc:hsqldb:hsqls://db.admc.com:9001/lm2
#username BLAINE
#password asecret
#truststore ${user.home}/ca/db/db-trust.store



# Template for a Postgresql database
#urlid blainedb
#url jdbc:postgresql://idun.africawork.org/blainedb
#username blaine
#password losung1
#driver org.postgresql.Driver

# Template for a MySQL database.  MySQL has poor JDBC support.
#urlid mysql-testdb
#url jdbc:mysql://hostname:3306/dbname
#username root
#password hiddenpwd
#driver com.mysql.jdbc.Driver

# Note that "databases" in SQL Server and Sybase are traditionally used for
# the same purpose as "schemas" with more SQL-compliant databases.

# Template for a Microsoft SQL Server database
#urlid msprojsvr
#url jdbc:microsoft:sqlserver://hostname;DatabaseName=DbName;SelectMethod=Cursor
# The SelectMethod setting is required to do more than one thing on a JDBC
# session (I guess Microsoft thought nobody would really use Java for
# anything other than a "hello world" program).
# This is for Microsoft's SQL Server 2000 driver (requires mssqlserver.jar
# and msutil.jar).
#driver com.microsoft.jdbc.sqlserver.SQLServerDriver
#username myuser
#password hiddenpwd

# Template for a Sybase database
#urlid sybase
#url jdbc:sybase:Tds:hostname:4100/dbname
#username blaine
#password hiddenpwd
# This is for the jConnect driver (requires jconn3.jar).
#driver com.sybase.jdbc3.jdbc.SybDriver

# Template for Embedded Derby / Java DB.
#urlid derby1
```

```
#url jdbc:derby:path/to/derby/directory;create=true
#username ${user.name}
#password any_noauthbydefault
#driver org.apache.derby.jdbc.EmbeddedDriver
# The embedded Derby driver requires derby.jar.
# There'a also the org.apache.derby.jdbc.ClientDriver driver with URL
# like jdbc:derby://<server>[:<port>]/databaseName, which requires
# derbyclient.jar.
# You can use \= to commit, since the Derby team decided (why???)
# not to implement the SQL standard statement "commit"!!
# Note that SqlTool can not shut down an embedded Derby database properly,
# since that requires an additional SQL connection just for that purpose.
# However, I've never lost data by not shutting it down properly.
# Other than not supporting this quirk of Derby, SqlTool is miles ahead of ij.
```

We will be using the "localhost-sa" sample urlid definition from the config file. The JDBC URL for this urlid is
jdbc:hsqldb:hsql://localhost. That is the URL for the default catalog of a HyperSQL Listener running
on the default port of the local host. You can read about URLs to connect to other catalogs with and without listeners
in other chapters of this guide.

Run SqlTool.

```
    java -jar path/to/sqltool.jar localhost-sa
```

If you get a prompt, then all is well. If security is of any concern to you at all, then you should change the privileged
password in the database. Use the command   SET PASSWORD   command to change SA's password.

```
    SET PASSWORD 'newpassword';
```

Set a *strong* password!

> **Note**
>
> If, like most UNIX System Administrators, you often need to make up strong passwords, I highly suggest
> the great little program pwgen [https://sourceforge.net/projects/pwgen/] . You
> can probably get it where you get your other OS packages. The command pwgen -1 is usually all
> you need.

Note that with SQL-conformant databases like HyperSQL 2.0, user names and passwords are case sensitive. If you don't
quote the name, it will be interpreted as upper-case, like any named SQL object. (Only for backwards compatibility,
we do make an exception for the special user name SA, but you should always use upper-case "SA" nevertheless).

When you're finished playing, exit with the command \q.

If you changed the SA password, then you need to update the password in the sqltool.rc file accordingly.

You can, of course, also access the database with any JDBC client program. You will need to modify your classpath
to include hsqldb.jar as well as your client class(es). You can also use the other HSQLDB client programs, such
as org.hsqldb.util.DatabasManagerSwing, a graphical client with a similar purpose to SqlTool.

You can use any normal UNIX account to run the JDBC clients, including SqlTool, as long as the account has read ac-
cess to the sqltool.jar file and to an sqltool.rc file. See the Utilities Guide about where to put sqltool.rc,
how to execute sql files, and other SqlTool features.

Create additional Accounts

Connect to the database as SA (or any other Administrative user) and run   CREATE USER   to create new accounts
for your catalog. HSQLDB accounts are database-catalog-specific, not Listener-specific.

In SQL-compliant databases, all database objects are created in a *schema*. If you don't specify a schema, then the new object will be created in the default schema. To create a database object, your account (the account that you connected with) must have the role DBA, or your account must have authorization for the target schema (see the CREATE SCHEMA command about this last). When you first create a HyperSQL catalog, it has only one database user-- SA, a DBA account, with an empty string password. You should set a password (as described above). You can create as many additional users as you wish. To make a user a DBA, you can use the "ADMIN" option to the CREATE USER command, command, or GRANT the DBA Role to the account after creating it.

Once an object is created, the object creator and users with the DBA role will have all privileges to work with that object. Other users will have only the rights which the pseudo-user PUBLIC has. To give specific users more permissions, even rights to read objects, you can GRANT permissions for specific objects, grant Roles (which encompass a set of permissions), or grant the DBA Role itself.

Since only people with a database account may do anything at all with the database, it is often useful to permit other database users to view the data in your tables. To optimize performance, reduce contention, and minimize administration, it is often best to grant SELECT to PUBLIC on table-like objects that need to be accessed by multiple database users, with the significant exception of any data which you want to keep secret. (Similary with EXECUTE priv for routines and USAGE priv for other object types). Note that this is not at all equivalent to giving the world or the Internet read access to your tables-- you are giving read access to people that have been given accounts for the target database catalog.

Shutdown

Do a clean database shutdown when you are finished with the database catalog. You need to connect up as SA or some other Admin user, of course. With SqlTool, you can run

```
    java -jar path/to/sqltool.jar --sql 'shutdown;' localhost-sa
```

You don't have to worry about stopping the Listener because it shuts down automatically when all served database catalogs are shut down.

Running Hsqldb as a System Daemon

You can, of course, run HSQLDB through inittab on System V UNIXes, but usually an init script is more convenient and manageable. This section explains how to set up and use our UNIX init script. Our init script is only for use by root. (That is not to say that the *Listener* will run as root-- it usually should not).

The main purpose of the init script is to start up a Listener for the database catalogs specified in your server.properties file; and to gracefully shut down these same catalogs. For each catalog defined by a server.database.X setting in your .properties file, you must define an administrative "urlid" in your sqltool.rc (these are used to access the catalogs for validation and shutdown purposes). Finally, you list the urlid names in your init script config file. If, due to firewall issues, you want to run a WebServer instead of a Server, then make sure you have a healthy WebServer with a webserver.properties set up, adjust your URLs in sqltool.rc, and set TARGET_CLASS in the config file.

By following the commented examples in the config file, you can start up any number of Server and/or WebServer listener instances with or without TLS encryption, and each listener instance can serve any number of HyperSQL catalogs (independent data sets), all with optimal efficiency from a single JVM process. There are instructions in the init script itself about how to run multiple, independently-configured JVM processes. Most UNIX installations, however, will run a single JVM with a single Listener instance which serves multiple catalogs, for easier management and more efficient resource usage.

After you have the init script set up, root can use it anytime to start or stop HSQLDB. (I.e., not just at system bootup or shutdown).

Portability of `hsqldb` init script

The primary design criterion of the init script is portability. It does not print pretty color startup/shutdown messages as is common in late-model Linuxes and HPUX; and it does not keep subsystem state files or use the startup/shutdown functions supplied by many UNIXes, because these features are all non-portable.

Offsetting these limitations, this one script does it's intended job great on the UNIX varieties I have tested, and can easily be modified to accommodate other UNIXes. While you don't have tight integration with OS-specific daemon administration guis, etc., you do have a well tested and well behaved script that gives good, utilitarian feedback.

Init script Setup Procedure

The strategy taken here is to get the init script to run your single Server or WebServer first (as specified by TARGET_CLASS). After that's working, you can customize the JVM that is run by running additional Listener instances in it, running your own application in it (embedding), or even overriding HSQLDB behavior with your own overriding classes.

1. Copy the init script `sample/hsqldb.init` to `hsqldb` in the directory where init scripts live on your variety of UNIX. The most common locations are `/etc/init.d` or `/etc/rc.d/init.d` on System V style UNIXes, `/usr/local/etc/rc.d` on BSD style UNIXes, and `/Library/StartupItems/hsqldb` on OS X (you'll need to create the directory for the last).

2. View your `server.properties` file. Make a note of every catalog define by a `server.database.X` setting. A couple steps down, you will need to set up administrative access for each of these catalogs. If you are using our sample `server.properties` file, you will just need to set up access for the catalog specified with `file:db0/dbo`.

> **Note**
>
> Pre-2.0 versions of the hsqldb init script required use of .properties settings of the form`server.urlid.X`. These settings are obsolete and should be removed.

3. Either copy HSQLDB_OWNER's `sqltool.rc` file into root's home directory, or set the value of `AUTH_FILE` to the absolute path of HSQLDB_OWNER's `sqltool.rc` file. This file is read directly by root, even if you run hsqldb as non-root (by setting HSQLDB_OWNER in the config file). If you copy the file, make sure to use `chmod` to restrict permissions on the new copy. The init script will abort with an appropriate exhortation if you have the permissions set incorrectly.

   You need to set up a urlid stanza in your `sqltool.rc` file for network access (i.e. JDBC URL with hsql:, hsqls:, http:, or https:) for each catalog in your `server.properties` file. For our example, you need to define a stanza for the `file:db0/db0` catalog. You must supply for this catalog, a hsql: JDBC URL, an administrative user name, and the password.

   Example 14.1. example sqltool.rc stanza

   ```
   urlid localhostdb1
   url jdbc:hsqldb:hsql://localhost
   username SA
   password secret
   ```

4. Look at the comment towards the top of the init script which lists recommended locations for the configuration file for various UNIX platforms. Copy the sample config file `sample/hsqldb.cfg` to one of the listed locations (your choice). Edit the config file according to the instructions in it. For our example, you will set the value of URLIDS to `localhostdb1`, since that is the urlid name that we used in the `sqltool.rc` file.

   ```
   # $Id: hsqldb.cfg 3583 2010-05-16 01:49:52Z unsaved $
   ```

```
# Sample configuration file for HyperSQL Server Listener.
# See the "HyperSQL on UNIX" chapter of the HyperSQL User Guide.

# N.b.!!!!  You must place this in the right location for your type of UNIX.
# See the init script "hsqldb" to see where this must be placed and
# what it should be renamed to.

# This file is "sourced" by a Bourne shell, so use Bourne shell syntax.

# This file WILL NOT WORK until you set (at least) the non-commented
# variables to the appropriate values for your system.
# Life will be easier if you avoid all filepaths with spaces or any other
# funny characters.  Don't ask for support if you ignore this advice.

# The URLIDS setting below is new and REQUIRED.  This setting replaces the
# server.urlid.X settings which used to be needed in your Server's
# properties file.

# -- Blaine (blaine dot simpson at admc dot com)

JAVA_EXECUTABLE=/usr/bin/java

# Unless you copied the jar files from another system, this typically
# resides at $HSQLDB_HOME/lib/sqltool.jar, where $HSQLDB_HOME is your HSQLDB
# software base directory.
# The file name may actually have a version label in it, like
# sqltool-1.2.3.jar (in which case, you must specify the full name here).
# A 'hsqldb.jar' file (with or without version label) must reside in the same
# directory as the specified sqltool.jar file.
SQLTOOL_JAR_PATH=/opt/hsqldb-2.0.0/hsqldb/lib/sqltool.jar
# For the sample value above, there must also exist a file
# /opt/hsqldb-2.0.0/hsqldb/lib/hsqldb*.jar.

# Where the file "server.properties" or "webserver.properties" resides.
SERVER_HOME=/opt/hsqldb-2.0.0/hsqldb/data

# What UNIX user the server will run as.
# (The shutdown client is always run as root or the invoker of the init script).
# Runs as root by default, but you should take the time to set database file
# ownerships to another user and set that user name here.
HSQLDB_OWNER=hsqldb

# The HSQLDB jar file specified in HSQLDB_JAR_PATH above will automatically
# be in the class path.  This arg specifies additional classpath elements.
# To embed your own application, add your jar file(s) or class base
# directories here, and add your main class to the INVOC_ADDL_ARGS setting
# below.  Another common use-case for adding to your class path is to make
# classes available to the DB engines for SQL/JRT functions and procedures.
#SERVER_ADDL_CLASSPATH=/usr/local/dist/currencybank.jar

# For startup or shutdown failures, you can save a lot of debugging time by
# temporarily adjusting down MAX_START_SECS and MAX_TERMINATE_SECS to a
# little over what it should take for successful startup and shutdown on
# your system.

# We require all Server/WebServer instances to be accessible within
# $MAX_START_SECS from when the Server/WebServer is started.
# Defaults to 60.
# Raise this is you are running lots of DB instances or have a slow server.
#MAX_START_SECS=200

# Max time to allow for JVM to die after all HSQLDB instances stopped.
# Defaults to 60.  Set high because the script will always continue as soon as
# the process has stopped.  The importance of this setting is, how long until
# a non-stopping-JVM-problem will be detected.
#MAX_TERMINATE_SECS=0
```

```
# NEW AND IMPORTANT!!!
# As noted at the top of this file, this setting replaces the old property
# settings server.urlid.X.
# Simply list the URLIDs for all DB instances which your *Server starts.
# Usually, these will exactly mirror the server.database.X settings in your
# server.properties or webserver.properties file.
# Each urlid listed here must be defined to a NETWORK url with Admin privileges
# in the AUTH_FILE specified below.  (Network type because we use this for
# inter-process communication)
# Separate multiple values with white space.  NO OTHER SPECIAL CHARACTERS!
# Make sure to quote the entire value if it contains white space separator(s).
URLIDS='localhostdb1'

# These are urlids # ** IN ADDITION TO URLIDS **, for instances which the init
# script should stop but not start.
# Most users will not need this setting.  If you need it, you'll know it.
# Defaults to none (i.e., only URLIDS will be stopped).
#SHUTDOWN_URLIDS='ondemand'

# SqlTool authentication file used only for shutdown.
# The default value will be sqltool.rc in root's home directory, since it is
# root who runs the init script.
# (See the SqlTool chapter of the HyperSQL Utilities Guide if you don't
# understand this).
#AUTH_FILE=/home/blaine/sqltool.rc

# Typical users will leave this unset and it will default to
# org.hsqldb.server.Server.  If you need to run the HSQLDB WebServer class
# instead, due to a firewall or routing impediment, set this to
# org.hsqldb.server.WebServer, see the docs about running WebServr, and
# set up a "webserver.properties" file instead of a "server.properties".
# The JVM that is started can invoke many classes (see the following item
# about that), but this is the server that is used (1) to check status,
# (2) to shut down the JVM.
#TARGET_CLASS=org.hsqldb.server.WebServer

# This is where you may specify both command-line parameters to TARGET_CLASS,
# plus any number of additional progams to run (along with their command-line
# parameters).  The MainInvoker program is used to embed these multiple
# static main invocations into a single JVM, so see the API spec for
# org.hsqldb.util.MainInvoker if you want to learn more.
# N.b. You should only use this setting to set HSQLDB Server or WebServer
# parameters if you run multiple instances of this class, since you can use the
# server/webserver.properties file for a single instance.
# Every additional class (in addition to the TARGET_CLASS)
# must be preceded with an empty string, so that MainInvoker will know
# you are giving a class name.  MainInvoker will invoke the normal
# static main(String[]) method of each such class.
# By default, MainInvoker will just run TARGET_CLASS with no args.
# Example that runs just the TARGET_CLASS with the specified arguments:
#INVOC_ADDL_ARGS='-silent false'   #but use server.properties property instead!
# Example that runs the TARGET_CLASS plus a WebServer:
#INVOC_ADDL_ARGS='"" org.hsqldb.server.WebServer'
# Note the empty string preceding the class name.
# Example that starts TARGET_CLASS with an argument + a WebServer +
# your own application with its args (i.e., the HSQLDB Servers are
# "embedded" in your application).  (Set SERVER_ADDL_CLASSPATH too).:
#INVOC_ADDL_ARGS='-silent false "" org.hsqldb.server.WebServer "" com.acme.Stone --env prod
 localhost'
#    but use server.properties for -silent option instead!
# Example to run a non-TLS server in same JVM with a TLS server.  In this
# case, TARGET_CLASS is Server which will run both in TLS mode by virtue of
# setting the tls, keyStore, and keyStorePassword settings in
# server*.properties, as described below; plus an "additional" Server with
# overridden 'tls' and 'port' settings:
#INVOC_ADDL_ARGS="'' org.hsqldb.server.Server --port 9002 --tls false"
```

```
# This is an important use case.  If you run more than one Server instance,
# you can specify different parameters for each here, even though only one
# server.properties file is supported.
# Note that you use nested quotes to group arguments and to specify the
# empty-string delimiter.

# The TLS_* settings have been obsoleted.
# To get your server running with TLS, set
# system.javax.net.ssl.keyStore=/path/to/your/private.keystore
# system.javax.net.ssl.keyStorePassword=secretPassword
# server.ssl=true
# IN server.properties or webserver.properties, and
# MAKE THE FILE OWNER-READ-ONLY!
# See the TLS Encryption section of the HyperSQL User Guide, paying attention
# to the security warning(s).
# If you are running with a private server cert, then you will also need to
# set "truststore" in the your SqlTool config file (location is set by the
# AUTH_FILE variable in this file, or it must be at the default location for
# HSQLDB_OWNER).

# Any JVM args for the invocation of the JDBC client used to verify DB
# instances and to shut them down (SqlToolSprayer).
# Server-side System Properties should normally be set with system.*
# settings in the server/webserver.properties file.
# This example specifies the location of a private trust store for TLS
# encryption.
# For multiple args, put quotes around entire value.
# If you are starting just a TLS_encrypted Listener, you need to uncomment
# this so the init scripts uses TLS to connect.
# If using a private keystore, you also need to set "truststore" settings in
# the sqltool.rc file.
#CLIENT_JVMARGS=-Djavax.net.debug=ssl
# This sample value displays useful debugging information about TLS/SSL.

# Any JVM args for the server.
# For multiple args, put quotes around entire value.
#SERVER_JVMARGS=-Xmx512m
# You can set the "javax.net.debug" property on the server side here, in the
# same exact way as shown for the client side above.
```

Verify that the init script works.

Just run

```
/path/to/hsqldb
```

as root to see the arguments you may use. Notice that you can run

```
/path/to/hsqldb status
```

at any time to see whether your HSQLDB Listener is running.

Re-run the script with each of the possible arguments to really test it good. If anything doesn't work right, then see the Troubleshooting the Init Script section.

5. Tell your OS to run the init script upon system startup and shutdown. If you are using a UNIX variant that has /etc/rc.conf or /etc/rc.conf.local (like BSD variants and Gentoo), you must set "hsqldb_enable" to "YES" in either of those files. (Just run cd /etc; ls rc.conf rc.conf.local to see if you have one of these files). For good UNIXes that use System V style init, you must set up hard links or soft links either manually or with management tools (such as chkconfig or insserv) or Gui's (like run level editors).

This paragraph is for Mac OS X users only. If you followed the instructions above, your init script should reside at /Library/StartupItems/hsqldb/hsqldb. Now copy the file StartupParameters.plist from

the directory `src/org.hsqldb/sample` of your HSQLDB distribution to the same directory as the init script. As long as these two files reside in `/Library/StartupItems/hsqldb`, your init script is active (for portability reasons, it doesn't check for a setting in `/etc/hostconfig`). You can run it as a *Startup Item* by running

```
SystemStarter {start|stop|restart} Hsqldb
```

Hsqldb is the service name. See the man page for `SystemStarter`. To disable the init script, wipe out the `/Library/StartupItems/hsqldb` directory. Hard to believe, but the Mac people tell me that during system shutdown the Startup Items don't run at all. Therefore, if you don't want your data corrupted, make sure to run "SystemStarter stop Hsqldb" before shutting down your Mac.

Follow the examples in the config file to add additional classes to the server JVM's classpath and to execute additional classes in your JVM. (See the `SERVER_ADDL_CLASSPATH` and `INVOC_ADDL_ARGS` items).

Troubleshooting the Init Script

Definitely look at the init script log file, which is at an OS-sependent location, but is usually at `/var/log/hsqldb.log`.

Do a `ps` to look for processes containing the string `hsqldb`, and try to connect to the database from any client. If the init script starts up your database successfully, but incorrectly reports that it has not, then your problem is with specification of urlid(s) or SqlTool setup. If your database really did not start, then skip to the next paragraph. Verify that your config file assigns a urlid for each catalog defined in `server.properties` or `webserver.properties`, then verify that you can run `SqlTool` as root to connect to the catalogs with these urlids. (For the latter test, use the `--rcfile` switch if you are setting `AUTH_FILE` in the init script config file).

If your database really is not starting, then verify that you can `su` to the database owner account and start the database. The command `su USERNAME -c ...` won't work on most UNIXes unless the target user has a real login shell. Therefore, if you try to tighten up security by disabling this user's login shell, you will break the init script. If these possibilities don't pan out, then debug the init script or seek help, as described below.

To debug the init script, run it in verbose mode to see exactly what is happening (and perhaps manually run the steps that are suspect). To run an init script (in fact, any sh shell script) in verbose mode, use `sh` with the `-x` or `-v` switch, like

```
sh -x path/to/hsqldb start
```

See the man page for `sh` if you don't know the difference between `-v` and `-x`.

If you want troubleshooting help, use the HSQLDB lists/forums. Make sure to include the revision number from your `hsqldb` init script (it's towards the top in the line that starts like "# $Id:"), and the output of a run of

```
sh -x path/to/hsqldb start > /tmp/hstart.log 2>&1
```

Upgrading

This section is for users who are using our UNIX init script, and who are upgrading their HyperSQL installation.

Most users will not have customized the init script itself, and your customizations will all be encapsulated in the init script configuration file. These users should just overwrite their init script with a new one from the HyperSQL installation, and manually merge config file settings. First, just copy the file `/sample/hsqldb.init` over top of of your init script (wherever it runs from). Then update your old config file according to the instructions in the new config file template at `sample/hsqldb.cfg`. You will have to change very few settings. If you are upgrading from a pre-2.0 installation to a post-2.0 installation, you will need to (1) add the setting `URLIDS`, as described above and in the inline comments, and (2) replace variable `HSQLDB_JAR_PATH` with `SQLTOOL_JAR_PATH` which (if you haven't guessed) should be set to the path to your `sqltool.jar` file.

Users who customized their init script will need to merge their customizations into the new init script.

Appendix A. Lists of Keywords

List of SQL Keywords
Fred Toussi
$Revision: 847 $
Published $Date: 2009-01-19 22:24:49 +0000 (Mon, 19 Jan 2009) $

List of SQL Standard Keywords

According to the SQL Standard, the SQL Language keywords cannot be used as identifiers (names of database objects such as columns and tables). HyperSQL has two modes of operation, which are selected with the SET DATABASE SQL NAMES { TRUE | FALSE } to allow or disallow the keywords as identifiers. The default mode is FALSE and allows the use of most keywords as identifiers. Even in this mode, keywords cannot be used as USER or ROLE identifiers.

ABS ALL ALLOCATE ALTER AND ANY ARE ARRAY AS ASENSITIVE ASYMMETRIC AT ATOMIC AUTHORIZATION AVG

BEGIN BETWEEN BIGINT BINARY BLOB BOOLEAN BOTH BY

CALL CALLED CARDINALITY CASCADED CASE CAST CEIL CEILING CHAR CHAR_LENGTH CHARACTER CHARACTER_LENGTH CHECK CLOB CLOSE COALESCE COLLATE COLLECT COLUMN COMMIT COMPARABLE CONDITION CONNECT CONSTRAINT CONVERT CORR CORRESPONDING COUNT COVAR_POP COVAR_SAMP CREATE CROSS CUBE CUME_DIST CURRENT CURRENT_CATALOG CURRENT_DATE CURRENT_DEFAULT_TRANSFORM_GROUP CURRENT_PATH CURRENT_ROLE CURRENT_SCHEMA CURRENT_TIME CURRENT_TIMESTAMP CURRENT_TRANSFORM_GROUP_FOR_TYPE CURRENT_USER CURSOR CYCLE

DATE DAY DEALLOCATE DEC DECIMAL DECLARE DEFAULT DELETE DENSE_RANK DEREF DESCRIBE DETERMINISTIC DISCONNECT DISTINCT DO DOUBLE DROP DYNAMIC

EACH ELEMENT ELSE ELSEIF END END_EXEC ESCAPE EVERY EXCEPT EXEC EXECUTE EXISTS EXIT EXP EXTERNAL EXTRACT

FALSE FETCH FILTER FIRST_VALUE FLOAT FLOOR FOR FOREIGN FREE FROM FULL FUNCTION FUSION

GET GLOBAL GRANT GROUP GROUPING

HANDLER HAVING HOLD HOUR

IDENTITY IN INDICATOR INNER INOUT INSENSITIVE INSERT INT INTEGER INTERSECT INTERSECTION INTERVAL INTO IS ITERATE

JOIN

LAG

LANGUAGE LARGE LAST_VALUE LATERAL LEAD LEADING LEAVE LEFT LIKE LIKE_REGEX LN LOCAL LOCALTIME LOCALTIMESTAMP LOOP LOWER

MATCH MAX MAX_CARDINALITY MEMBER MERGE METHOD MIN MINUTE MOD MODIFIES MODULE MONTH MULTISET

NATIONAL NATURAL NCHAR NCLOB NEW NO NONE NORMALIZE NOT NTH_VALUE NTILE NULL NULLIF NUMERIC

OCCURRENCES_REGEX OCTET_LENGTH OF OFFSET OLD ON ONLY OPEN OR ORDER OUT OUTER OVER OVERLAPS OVERLAY

PARAMETER PARTITION PERCENT_RANK PERCENTILE_CONT PERCENTILE_DISC POSITION POSITION_REGEX POWER PRECISION PREPARE PRIMARY PROCEDURE

RANGE RANK READS REAL RECURSIVE REF REFERENCES REFERENCING REGR_AVGX REGR_AVGY REGR_COUNT REGR_INTERCEPT REGR_R2 REGR_SLOPE REGR_SXX REGR_SXY REGR_SYY RE-LEASE REPEAT RESIGNAL RESULT RETURN RETURNS REVOKE RIGHT ROLLBACK ROLLUP ROW ROW_NUMBER ROWS

SAVEPOINT SCOPE SCROLL SEARCH SECOND SELECT SENSITIVE SESSION_USER SET SIGNAL SIMILAR SMALLINT SOME SPECIFIC SPECIFICTYPE SQL SQLEXCEPTION SQLSTATE SQLWARN-ING SQRT STACKED START STATIC STDDEV_POP STDDEV_SAMP SUBMULTISET SUBSTRING SUBSTRING_REGEX SUM SYMMETRIC SYSTEM SYSTEM_USER

TABLE TABLESAMPLE THEN TIME TIMESTAMP TIMEZONE_HOUR TIMEZONE_MINUTE TO TRAILING TRANSLATE TRANSLATE_REGEX TRANSLATION TREAT TRIGGER TRIM TRIM_ARRAY TRUE TRUN-CATE

UESCAPE UNDO UNION UNIQUE UNKNOWN UNNEST UNTIL UPDATE UPPER USER USING

VALUE VALUES VAR_POP VAR_SAMP VARBINARY VARCHAR VARYING

WHEN WHENEVER WHERE WIDTH_BUCKET WINDOW WITH WITHIN WITHOUT WHILE

YEAR

List of SQL Keywords Disallowed as HyperSQL Identifiers

A subset of SQL Standard keywords cannot be used at all as HyperSQL identifiers. The keywords are as follows:

ADMIN AND ALL ANY AS AT AVG

BETWEEN BOTH BY

CALL CASE CAST COALESCE CORRESPONDING CONVERT COUNT CREATE CROSS

DISTINCT DROP

ELSE END EVERY EXISTS EXCEPT

FOR FROM FULL

GRANT GROUP

HAVING

IN INNER INTERSECT INTO IS

JOIN

LEFT LEADING LIKE

MAX MIN

NATURAL NOT NULLIF

ON ORDER OR OUTER

PRIMARY

REFERENCES RIGHT

SELECT SET SOME STDDEV_POP STDDEV_SAMP SUM

TABLE THEN TO TRAILING TRIGGER

UNION UNIQUE USING

VALUES VAR_POP VAR_SAMP

WHEN WHERE WITH

Appendix B. Building HyperSQL Jars

How to build customized or specialized jar files

Fred Toussi

$Revision: 3556 $

Published $Date: 2010-03-26 19:09:40 -0400 (Fri, 26 Mar 2010) $

Purpose

From 2.0, the supplied `hsqldb.jar` file is built with Java 1.6. If you want to run with a 1.5 or older JVM, or if you want to use an alternative jar (`hsqldb-min.jar`, etc.) you must build the desired jar with a Java JDK and Ant version 1.7.

Building with Apache Ant

You should use version 1.7.x of Ant (Another Neat Tool) to do builds with HyperSQL.

Obtaining Ant

Ant is a part of the Jakarta/Apache Project.

- Home of the Apache Ant project   [http://ant.apache.org]

- The   Installing Ant   [http://ant.apache.org/manual/install.html#installing] page of the   Ant Manual   [http://ant.apache.org/manual]. Follow the directions for your platform.

Building Hsqldb with Ant

Once you have unpacked the zip package for hsqldb, under the `/hsqldb` folder, in `/build` there is a `build.xml` file that builds the `hsqldb.jar` with Ant (Ant must be already installed). To use it, change to `/build` then type:

```
ant -projecthelp
```

This displays the available ant targets, which you can supply as command line arguments to ant. These include

hsqldb	to build the `hsqldb.jar` file
explainjars	Lists all targets which build jar files, with an explanation of the purposes of the different jars.
clean	to clean up the /classes directory that is created
clean-all	to remove the old jar and doc files as well
javadoc	to build javadoc
hsqldbmain	to build a smaller jar for HSQLDB that does not contain utilities
hsqljdbc	to build an extremely small jar containing only the client-side JDBC driver (can connect only to a HyperSQL Server).
hsqldbmin	to build a small jar that supports *in-process* catalogs, but neither running nor connecting to Hyper-SQL Servers.
sqltool	to build sqltool.jar, which contains only the SqlTool classes.

...   Many more targets are available. Run `ant -p` and `ant explainjars`.

HSQLDB can be built in any combination of two JRE (Java Runtime Environment) versions and many jar file sizes.

A jar built with an older JRE is compatible for use with a newer JRE (you can compile with Java 1.5 and run with 1.6). But the newer JDBC capabilities of the JRE will be not be available.

The client jar (`hsqljdbc.jar`) contains only the HSQLDB JDBC Driver client. The smallest engine jar (`hsqldbmin.jar`) contains the engine and the HSQLDB JDBC Driver client. The default size (`hsqldb.jar`) also contains server mode support and the utilities. The largest size (`hsqldbtest.jar`)includes some test classes as well. Before building the `hsqldbtest.jar` package, you should download the junit jar from http://www.junit.org and put it in the `/lib` directory, alongside `servlet.jar`, which is included in the .zip package.

If you want your code built for high performance, as opposed to debugging (in the same way that we make our production distributions), make a file named `build.properties` in your build directory with the contents

```
build.debug: false
```

The resulting Java binaries will be faster and smaller, at the cost of exception stack traces not identifying source code locations (which can be extremely useful for debugging).

After installing Ant on your system use the following command from the `/build` directory. Just run `ant explainjars` for a concise list of all available jar files.

```
ant explainjars
```

The command displays a list of different options for building different sizes of the HSQLDB Jar. The default is built using:

Example B.1. Buiding the standard Hsqldb jar file with Ant

```
ant hsqldb
```

The Ant method always builds a jar with the JDK that is used by Ant and specified in its JAVA_HOME environment variable.

Building for Older JDKs

HyperSQL version 2.0 cannot be directly compiled or used with JDK 1.4. It may be possible to use the RetroTranslator tool to achieve this. The suggested procedure is as follows: First use Ant with JDK 1.5 and build the jar. Then translate the jar using RetroTranslator with backport (which bundles replacement classes for concurrency control). This translation should cover the concurrency features that are specific to version 1.5 and later.

```
ant switchtojdk14
ant hsqldb
-- translate the jar
```

Building with IDE's

All HyperSQL source files are supplied ready to compile. There is no complex pre-compile stage. It is therefore possible to compile the sources with an IDE, without using ant. Only if compilation with Java 1.5 is required, you should first run the Ant code switcher task before compiling and remove from the source directories a few source files that are specific to Java 6 (these are listed in the build.xml file).

Hsqldb CodeSwitcher

CodeSwitcher is a tool to manage different version of Java source code. It allows to compile HyperSQL for different JDKs. It is something like a precompiler in C but it works directly on the source code and does not create intermediate output or extra files.

CodeSwitcher is used internally in the Ant build. You do not have to use it separately to compile HyperSQL.

CodeSwitcher reads the source code of a file, removes comments where appropriate and comments out the blocks that are not used for a particular version of the file. This operation is done for all files of a defined directory, and all subdirectories.

Example B.2. Example source code before CodeSwitcher is run

```
    ...

//#ifdef JAVA2

    properties.store(out,"hsqldb database");

//#else

/*

    properties.save(out,"hsqldb database");

*/

//#endif

    ...
```

The next step is to run CodeSwitcher.

Example B.3. CodeSwitcher command line invocation

```
java org.hsqldb.util.CodeSwitcher . -JAVA2
```

The '.' means the program works on the current directory (all subdirectories are processed recursively). -JAVA2 means the code labelled with JAVA2 must be switched off.

Example B.4. Source code after CodeSwitcher processing

```
    ...

//#ifdef JAVA2

/*

    pProperties.store(out,"hsqldb database");

*/

//#else

    pProperties.save(out,"hsqldb database");

//#endif
```

```
        ...
```

For detailed information on the command line options run `java org.hsqldb.util.CodeSwitcher`. Usage examples can be found in the build.xml file in the `/build` directory.

Building documentation

The JavaDoc can be built simply by invoking the javadoc target.

The two Guides are in DocBook XML source format. To rebuild, run the Ant target `gen-docs`. Instructions will be displayed. See the file `doc-src/readme-docauthors.txt` for tips.

Appendix C. HyperSQL with OpenOffice.org

How to use HyperSQL with OpenOffice.org

Fred Toussi

$Revision: 3498 $

Published $Date: 2010-03-06 12:42:28 -0500 (Sat, 06 Mar 2010) $

HyperSQL with OpenOffice.org

OpenOffice.org includes HyperSQL and uses it for embedded databases. Our collaboration with OpenOffice.org developers over the last few years has benefited the development and maturity of HyperSQL. Before integration into OOo, HSQLDB was intended solely for application-specific database access. The application developer was expected to resolve any integration issues. Because OpenOffice.org is used by a vast range of users, from schoolchildren to corporate developers, a much higher level of quality assurance has been required and we have achieved it with constant help and feedback from OOo users and developers.

Apart from embedded use, you may want to use OpenOffic.org with a HyperSQL server instance. The typical use for this is to allow multiple office users accessing the same database. There is, however, a strong case for using OOo to develop your database schema and application, even if the database is intended for your own application.

Using OpenOffice.org as a Database Tool

OpenOffice.org is a very powerful database front end. If you want to create schemas, edit tables, edit the database contents manually, design and produce well-formatted reports, then OpenOffice.org is probably the best open source tools currently available.

To connect from OpenOffice.org to your database, first run a local server instance for the database. This is describes in the Network Listeners chapter of this guide.

When you connect from OpenOffice.org, you must specify connection to an external database and use the URL property "default_schema=true". For example, the URL to connect the local database may be like

```
jdbc;hsqldb:hsql://localhost/mydb;default_schema=true
```

The only current limitation is that OpenOffice.org only works with the PUBLIC schema. This limitation will hopefully removed in the future versions of OOo.

When using of HyperSQL with OOo, you must use the HyperSQL jar that is supplied with OOo. This wil hopefuly be a version 2.0 jar in the future versions of OOo.

Converting .odb files to use with HyperSQL Server

You may already have an OOo database file, which you want to use outside OOo, or as a server database. The file is in fact in the standard ZIP format and contains the normal HyperSQL database files. Just use a utility such as 7Zip to expand the .odb file. In the /db directory, there are files such as .script, .data, etc. Just rename these files into mydb.script, mydb.data, etc. You can now open the mydb database directly with HyperSQL as an embedded database or as a server instance.

Appendix D. HyperSQL File Links

HyperSQL Files referred to in this Guide

HyperSQL files referred to in the text may be retrieved from the canonical HyperSQL documentation site, http://hsqldb.org/doc/2.0, or from the same location you are reading this page from.

> **Note**
>
> If you are reading this document with a standalone PDF reader, only the http://hsqldb.org/doc/2.0/... links will function.

Pairs of local + http://hsqldb.org/doc/2.0 links for referenced files.

- Local: ../apidocs/org/hsqldb/jdbc/JDBCConnection.html

  http://hsqldb.org/doc/2.0/apidocs/org/hsqldb/jdbc/JDBCConnection.html

- Local: ../apidocs/org/hsqldb/jdbc/JDBCDriver.html

  http://hsqldb.org/doc/2.0/apidocs/org/hsqldb/jdbc/JDBCDriver.html

- Local: ../apidocs/org/hsqldb/jdbc/JDBCDatabaseMetaData.html

  http://hsqldb.org/doc/2.0/apidocs/org/hsqldb/jdbc/JDBCDatabaseMetaData.html

- Local: ../apidocs/org/hsqldb/jdbc/JDBCResultSet.html

  http://hsqldb.org/doc/2.0/apidocs/org/hsqldb/jdbc/JDBCResultSet.html

- Local: ../apidocs/org/hsqldb/jdbc/JDBCStatement.html

  http://hsqldb.org/doc/2.0/apidocs/org/hsqldb/jdbc/JDBCStatement.html

- Local: ../apidocs/org/hsqldb/jdbc/JDBCPreparedStatement.html

  http://hsqldb.org/doc/2.0/apidocs/org/hsqldb/jdbc/JDBCPreparedStatement.html

- Local: ../apidocs/org/hsqldb/util/MainInvoker.html

  http://hsqldb.org/doc/2.0/apidocs/org/hsqldb/util/MainInvoker.html

- Local: ../apidocs/index.html

  http://hsqldb.org/doc/2.0/apidocs/

- Local: ../verbatim/src/org/hsqldb/server/Servlet.java

  http://hsqldb.org/doc/2.0/verbatim/src/org/hsqldb/server/Servlet.java

- Local: ../verbatim/src/org/hsqldb/Tokens.java

  http://hsqldb.org/doc/2.0/verbatim/src/org/hsqldb/Tokens.java

- Local: ../verbatim/src/org/hsqldb/server/WebServer.java

  http://hsqldb.org/doc/2.0/verbatim/src/org/hsqldb/server/WebServer.java

- Local: ../verbatim/src/org/hsqldb/test/TestBase.java

  http://hsqldb.org/doc/2.0/verbatim/src/org/hsqldb/test/TestBase.java

- Local: ../verbatim/src/org/hsqldb/Trigger.java

  http://hsqldb.org/doc/2.0/verbatim/src/org/hsqldb/Trigger.java

- Local: ../verbatim/src/org/hsqldb/sample/TriggerSample.java

  http://hsqldb.org/doc/2.0/verbatim/src/org/hsqldb/test/sample/TriggerSample.java

- Local: ../verbatim/src/org/hsqldb/util/MainInvoker.java

  http://hsqldb.org/doc/2.0/verbatim/src/org/hsqldb/util/MainInvoker.java

- Local: ../verbatim/sample/hsqldb.cfg

  http://hsqldb.org/doc/2.0/verbatim/sample/hsqldb.cfg

- Local: ../verbatim/sample/acl.txt

  http://hsqldb.org/doc/2.0/verbatim/sample/acl.txt

- Local: ../verbatim/sample/server.properties

  http://hsqldb.org/doc/2.0/verbatim/sample/server.properties

- Local: ../verbatim/sample/sqltool.rc

  http://hsqldb.org/doc/2.0/verbatim/sample/sqltool.rc

- Local: ../verbatim/sample/hsqldb.init

  http://hsqldb.org/doc/2.0/verbatim/sample/hsqldb.init

SQL Index

Symbols

General Index

Symbols

_SYSTEM ROLE, 76

A

ABS function, 147
ACL, 194
ACOS function, 147
ADD COLUMN, 53
ADD CONSTRAINT, 55
ADD DOMAIN CONSTRAINT, 57
aggregate function, 102
ALL and ANY predicates, 99
ALTER COLUMN, 53
alter column nullability, 54
ALTER DOMAIN, 56
alter identity column, 54
ALTER routine, 60
ALTER SEQUENCE, 62
ALTER TABLE, 53
ALTER USER ... SET INITIAL SCHEMA, 79
ALTER USER ... SET PASSWORD, 78
ALTER view, 56
Ant, 213
ASCII function, 143
ASIN function, 147
ATAN2 function, 147
ATAN function, 147
AUTHORIZATION IDENTIFIER, 76

B

backup, 168
BACKUP DATABASE, 172
BETWEEN predicate, 97
binary literal, 87
BINARY types, 13
BIT_LENGTH function, 145
BITAND function, 147
bit literal, 87
BITOR function, 147
BIT types, 14
BITXOR function, 148
boolean literal, 88
BOOLEAN types, 12
boolean value expression, 96

C

CARDINALITY function, 155
CASCADE or RESTRICT, 45
case expression, 92
CASE WHEN in routines, 123

CAST, 93
CEIL function, 147
CHANGE_AUTHORIZATION, 77
CHARACTER_LENGTH, 145
character literal, 86
CHARACTER types, 12
character value function, 95
CHECK constraint, 51
CHECKPOINT, 172
COALESCE expression, 92
COALESCE function, 155
COLLATE, 102
column definition, 48
column reference, 90
COMMENT, 45
COMMIT, 34
comparison predicate, 97
CONCAT function, 143
CONSTRAINT, 102
CONSTRAINT (table constraint), 49
CONSTRAINT name and characteristics, 49
contextually typed value specification, 90
CONVERT function, 155
COS function, 148
COT function, 148
CREATE_SCHEMA ROLE, 77
CREATE AGGREGATE FUNCTION, 129
CREATE ASSERTION, 65
CREATE CAST, 64
CREATE CHARACTER SET, 64
CREATE COLLATION, 64
CREATE DOMAIN, 56
CREATE FUNCTION, 131
CREATE INDEX, 63
CREATE PROCEDURE, 131
CREATE ROLE, 79
CREATE SCHEMA, 46
CREATE SEQUENCE, 61
CREATE TABLE, 47
CREATE TRANSLATION, 65
CREATE TRIGGER, 57, 139
CREATE TYPE, 63
CREATE USER, 78
CREATE VIEW, 55
CROSS JOIN, 107
CRYPT_KEY, 173
CRYPT_KEY function, 157
CURDATE function, 151
CURRENT_CATALOG function, 158
CURRENT_DATE function, 151
CURRENT_ROLE function, 158
CURRENT_SCHEMA function, 158
CURRENT_TIME function, 151
CURRENT_TIMESTAMP function, 151