

# **Hsqldb User Guide**

**The HSQLDB Development Group**  
**Edited by Blaine Simpson and Fred Toussi**

---

# Hsqldb User Guide

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

Published \$Date: 2004/07/18 03:48:05 \$

Copyright 2002-2004 HSQLDB Development Group. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license.

---

---

---

---

# Table of Contents

Introduction .....	x
Available formats for this document .....	x
1. Running and Using Hsqldb .....	1
Introduction .....	1
Running Tools .....	1
Running Hsqldb .....	2
Server Modes .....	2
Hsqldb Server .....	3
Hsqldb Web Server .....	3
Hsqldb Servlet .....	3
In-Process (Standalone) Mode .....	4
Memory-Only Databases .....	5
General .....	5
Closing the Database .....	5
Using Multiple Databases in One JVM .....	5
Creating a New Database .....	5
Using the Database Engine .....	6
Different Types of Tables .....	6
Constraints and Indexes .....	7
SQL Support .....	7
JDBC Support .....	7
2. SQL Issues .....	8
Purpose .....	8
SQL Standard Support .....	8
Constraints and Indexes .....	8
Primary Key Constraints .....	8
Unique Constraints .....	8
Unique Indexes .....	9
FOREIGN KEYS .....	9
Indexes and Query Speed .....	9
Where Condition or Join .....	10
Subqueries and Joins .....	11
Types and Arithmetic Operations .....	11
Integral Types .....	12
Other Numeric Types .....	12
Bit and Boolean Types .....	13
Storage and Handling of Java Objects .....	13
Type Size, Precision and Scale .....	13
Sequences and Identity .....	14
Identity Auto-Increment Columns .....	14
Sequences .....	14
Issues with Transactions .....	15
New Features and Changes .....	15
3. UNIX Quick Start .....	16
Purpose .....	16
Installation .....	16
Setting up Database Instance and Server .....	18
Accessing your Database .....	19
Create additional Accounts .....	21
Shutdown .....	22
Running Hsqldb as a System Daemon .....	22
Portability of hsqldb init script .....	22
Init script Setup Procedure .....	23

---

Troubleshooting the Init Script .....	26
4. Advanced Topics .....	27
Purpose .....	27
Connections .....	27
Connection properties .....	28
Properties Files .....	29
Server and Web Server Properties .....	30
Starting a Server from your application .....	31
Individual Database Properties .....	31
JDBC Stream Based Methods .....	33
Managing Database Connections .....	34
Memory and Disk Use .....	34
Cache Memory Allocation .....	35
Upgrading and Managing Databases .....	36
Upgrading Using the SCRIPT Command .....	36
Manual Changes to the .script File .....	37
Backing Up Databases .....	37
5. Text Tables .....	38
The Implementation .....	38
Definition of Tables .....	38
Scope and Reassignment .....	39
Null Values in Columns of Text Tables .....	39
Configuration .....	39
Text File Issues .....	41
Text File Global Properties .....	41
Importing from a Text Table file .....	42
6. TLS .....	43
Requirements .....	43
Encrypting your JDBC connection .....	43
Client-Side .....	43
Server-Side .....	45
JSSE .....	45
Making a Private-key Keystore .....	45
CA-Signed Cert .....	46
Non-CA-Signed Cert .....	46
Automatic Server or WebServer startup on UNIX .....	46
7. SqlTool .....	47
Purpose .....	47
The Bare Minimum .....	47
Authentication Setup .....	49
Interactive .....	51
SQL History .....	57
Raw mode .....	57
Non-Interactive .....	57
Giving SQL on the Command Line .....	58
SQL Files .....	58
Optimally Compatible SQL Files .....	60
Comments .....	60
Special Commands and Buffer Commands in SQL Files .....	61
Automation .....	62
Getting Interactive Functionality with SQL Files .....	62
Character Encoding .....	63
Generating Text or HTML Reports .....	63
Procedural Language .....	64
Variables .....	65
Logical Expressions .....	66
Flow Control .....	66
Using hsqlsqltool.jar .....	68

---

Unit Testing SqlTool .....	69
8. SQL Syntax .....	70
Notational Conventions Used in this Chapter .....	70
SQL Commands .....	70
ALTER INDEX .....	70
ALTER SEQUENCE .....	70
ALTER TABLE .....	70
ALTER USER .....	72
CALL .....	72
CHECKPOINT .....	72
COMMIT .....	72
CONNECT .....	72
CREATE ALIAS .....	72
CREATE INDEX .....	73
CREATE SEQUENCE .....	73
CREATE TABLE .....	73
CREATE TRIGGER .....	75
CREATE USER .....	76
CREATE VIEW .....	76
DELETE .....	77
DISCONNECT .....	77
DROP INDEX .....	78
DROP SEQUENCE .....	78
DROP TABLE .....	78
DROP TRIGGER .....	78
DROP USER .....	78
DROP VIEW .....	79
GRANT .....	79
INSERT .....	79
REVOKE .....	79
ROLLBACK .....	79
SAVEPOINT .....	80
SCRIPT .....	80
SELECT .....	80
SET AUTOCOMMIT .....	81
SET IGNORECASE .....	81
SET LOGSIZE .....	82
SET PASSWORD .....	82
SET PROPERTY .....	82
SET REFERENTIAL INTEGRITY .....	82
SET SCRIPTFORMAT .....	83
SET TABLE INDEX .....	83
SET TABLE READONLY .....	83
SET TABLE SOURCE .....	83
SET WRITE DELAY .....	84
SHUTDOWN .....	84
UPDATE .....	85
Data Types .....	85
SQL Comments .....	86
Stored Procedures / Functions .....	86
Built-in Functions and Stored Procedures .....	87
SQL Expression .....	91
A. Building Hsqldb .....	94
Purpose .....	94
Building with Ant .....	94
Obtaining Ant .....	94
Building Hsqldb with Ant .....	94
Building with DOS Batch Files .....	95

Hsqldb CodeSwitcher .....	96
B. First JDBC Client Example .....	98
C. Hsqldb Database Files and Recovery .....	102
.....	102
States .....	102
.....	102
.....	102
.....	103
Procedures .....	103
Clean Shutdown .....	103
Startup .....	104
Repair .....	104
D. Running Hsqldb with OpenOffice.org .....	106
Introduction .....	106
Installing .....	106
Setting up OpenOffice.org .....	106
On Windows .....	106
On Linux .....	106
E. Hsqldb Test Utility .....	108
F. Database Manager .....	110
Brief Introduction .....	110
G. Transfer Tool .....	111
Brief Introduction .....	111

---

# List of Tables

- 1. Alternate formats of this document ..... x
- 4.1. Hsqldb URL Components ..... 27
- 4.2. Connection Properties ..... 28
- 4.3. Hsqldb Server Properties Files ..... 29
- 4.4. Property File Properties ..... 30
- 4.5. Server Property File Properties ..... 30
- 4.6. WebServer Property File Properties ..... 31
- 4.7. Database-specific Property File Properties ..... 31
- 8.1. Data Types ..... 85



---

## List of Examples

1.1. Java code to connect to the local Server above .....	4
2.1. Column values which satisfy a 2-column UNIQUE constraint .....	9
2.2. Query comparison .....	11
2.3. Numbering returned rows of a SELECT in sequential order .....	14
3.1. server.properties fragment .....	25
3.2. example sqltool.rc stanza .....	25
6.1. Exporting certificate from the server's keystore .....	44
6.2. Adding a certificate to the client keystore .....	44
6.3. Specifying your own trust store to a JDBC client .....	44
6.4. Running an Hsqldb server with TLS encryption .....	45
6.5. Getting a pem-style private key into a JKS keystore .....	46
7.1. Sample sqltool.rc File .....	49
7.2. Defining and using an alias (PL variable) .....	56
7.3. Piping input into SqlTool .....	57
7.4. Valid comment example .....	60
7.5. Invalid comment example .....	60
7.6. Simple SQL file using PL .....	65
7.7. SQL File showing use of most PL features .....	67
A.1. Building the standard Hsqldb jar file with Ant .....	95
A.2. Example source code before CodeSwitcher is run .....	96
A.3. CodeSwitcher command line invocation .....	96
A.4. Source code after CodeSwitcher processing .....	96
B.1. JDBC Client source code example .....	98

---

# Introduction

If you notice any mistakes in this document, please email the author listed at the beginning of the chapter. If you have problems with the procedures themselves, please use the HSQLDB support facilities which are listed at [http://sourceforge.net/support/getsupport.php?group\\_id=23316](http://sourceforge.net/support/getsupport.php?group_id=23316).

## Available formats for this document

This document is available in several formats.

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

<http://hsqldb.sourceforge.net/guide> 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.sourceforge.net/guide>.

Sometimes, distributions other than <http://hsqldb.sourceforge.net/guide> 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.sourceforge.net/guide>.

**Table 1. Alternate formats of this document**

format	your distro	at <a href="http://hsqldb.sourceforge.net/guide">http://hsqldb.sourceforge.net/guide</a>
Chunked HTML	index.html	<a href="http://hsqldb.sourceforge.net/guide/index.html">http://hsqldb.sourceforge.net/guide/index.html</a>
All-in-one HTML	guide.html	<a href="http://hsqldb.sourceforge.net/guide/guide.html">http://hsqldb.sourceforge.net/guide/guide.html</a>
PDF	guide.pdf	<a href="http://hsqldb.sourceforge.net/guide/guide.pdf">http://hsqldb.sourceforge.net/guide/guide.pdf</a>

---

# Chapter 1. Running and Using Hsqldb

Fred Toussi, HSQLDB Development Group <ft@cluedup.com>

Copyright 2002-2004 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 HSQLDB Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.  
\$Date: 2004/08/09 17:10:11 \$

## Introduction

The HSQLDB jar package is located in the /lib lirectory and contains several components and programs. Different commands are used to run each program.

### Components of the Hsqldb jar package

- HSQLDB RDBMS
- HSQLDB JDBC Driver
- Database Manager (Swing and AWT versions)
- Transfer Tool (AWT version)
- Query Tool (AWT)
- Sql Tool (command line)

The HSQLDB RDBMS and JDBC Driver provide the core functionality. The rest are general-purpose database tools that can be used with any database engine that has a JDBC driver.

## Running Tools

All tools can be run in the standard way for archived Java classes. In the following example the AWT version of the Database Manager, the `hsqldb.jar` is located in the directory `../lib` relative to the current directory.

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

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

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

### Main classes for the Hsqldb tools

- `org.hsqldb.util.DatabaseManager`

- `org.hsqldb.util.DatabaseManagerSwing`
- `org.hsqldb.util.Transfer`
- `org.hsqldb.util.QueryTool`
- `org.hsqldb.util.SqlTool`

Some tools, such as the Database Manager or SQL Tool, can use command line arguments or entirely rely on them. You can add the command line argument `-?` to get a list of available arguments for these tools. Database Manager and Transfer Tool feature a graphical user interface and can be explored interactively.

## Running Hsqldb

HSQLDB can be run in a number of different ways. In general these are divided into Server Modes and In-Process Mode (also called Standalone Mode). A different sub-program from the jar is used to run HSQLDB in each mode.

Each HSQLDB database 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`

The properties files contains general 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 zipped backup of the last known consistent state of the data file. All these files are essential and should never be deleted. If the database has no cached tables, the `test.data` and `test.backup` files will not be present. In addition to those files, HSQLDB database may link to any formatted text files, such as CSV lists, anywhere on the disk.

While the "test" database is operational, 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. In some circumstances, a `test.data.old` is created and deleted afterwards.

### 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.

## Server Modes

---

Server modes provide the maximum accessibility. The database engine runs in a JVM and listens for connections from programs on the same computer or other computers on the network. Several different programs can connect to the server and retrieve or update information. Applications programs (clients) connect to the server using the HSQLDB JDBC driver. In most server modes, the server serves up to 10 databases that are specified at the time of running the server.

Server modes can use preset properties or command line arguments as detailed in the Advanced Topics chapter. There are three server modes, based on the protocol used for communications between the client and server.

## Hsqldb 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.\*".

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

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

## Hsqldb Web Server

This mode is used when access to the computer hosting the database server is restricted to the HTTP protocol. The only reason for using the Web Server mode is restrictions imposed by firewalls on the client or server machines and it should not be used where there are no such restrictions. The HSQLDB Web Server is a special web server that allows JDBC clients to connect via HTTP. From 1.7.2 this mode also supports transactions.

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

```
org.hsqldb.WebServer
```

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

## Hsqldb Servlet

This uses the same protocol as the Web Server. 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 `hsqldbServlet` 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 `hsqldbServlet.java` to see the details.

Both Web 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.

## Connecting to a Database running as a Server

Once an HSQLDB 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 `JdbcConnection` [[./src/org/hsqldb/jdbc/jdbcConnection.html](#)] (located in the `/doc/src` 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 Server above

```
try {
    Class.forName("org.hsqldb.jdbcDriver" );
} catch (Exception e) {
    System.out.println("ERROR: failed to load HSQLDB JDBC driver.");
    e.printStackTrace();
    return;
}

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

In some circumstances, you may have to use the following line to get the driver.

```
Class.forName("org.hsqldb.jdbcDriver").newInstance();
```

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

## Security Considerations

When HSQLDB is run as server, the port should be adequately protected with a firewall. The password for the default system user should also be changed from the default empty string. When the server is run with the HTTP protocol, public access should not be allowed to confidential databases as it is not difficult to spoof an existing open session.

## In-Process (Standalone) Mode

This mode runs the database engine as part of your application program in the same Java Virtual Machine. For some applications this mode can be 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. In 1.7.2, you can run a server instance in a thread from the same virtual machine as your application and provide external access to your in-process database.

The recommended way of using the in-process mode in an application is to use an HSQLDB Server instance for the database while developing the application and then switch to In-Process mode for deployment.

An In-Process Mode database is started from JDBC, with the database file path specified in the connection URL. For example, if the 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",
```

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 Javadoc for `jdbcConnection` [[../src/org/hsqldb/jdbc/jdbcConnection.html](http://src/org/hsqldb/jdbc/jdbcConnection.html)] for more details.

## Memory-Only Databases

It is possible to run HSQLDB in a way that the database is not persistent and exists entirely in random access memory. As no information is written to disk, this mode should be used only for internal processing of application data, in applets or certain special applications. This mode is specified by the `mem:` protocol.

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

You can also run a memory-only server instance by specifying the same URL in the `server.properties`. This usage is not common and is limited to special applications where the database server is used only for exchanging information between clients, or for non-persistent data.

## General

### Closing the Database

All databases running in different modes can be closed with the `SHUTDOWN` command, issued as an SQL query. In 1.7.2, in-process databases are no longer closed when the last connection to the database is explicitly closed via JDBC, a `SHUTDOWN` is required. When `SHUTDOWN` is issued, all active transactions are rolled back. 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 size. This command should be issued periodically, especially when lots of inserts, updates or deletes are performed on the cached tables. Changes to the structure of the database, such as dropping or modifying tables or indexes also create large amounts of unused space that can be reclaimed using this command.

### Using Multiple Databases in One JVM

In the above examples each server serves only one database and only one in-memory database can be created. However, from version 1.7.2, HSQLDB can serve several databases in multiple server modes and allow simultaneous access to multiple in-process and memory-only databases. These capabilities are covered in the Advanced Topics chapter.

### 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.

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.

## Using the Database Engine

Once a connection is established to a database in any mode, JDBC methods are used to interact with the database. The Javadoc for `JdbcConnection` [[../src/org/hsqldb/jdbc/jdbcConnection.html](#)], `JdbcDriver` [[../src/org/hsqldb/jdbcDriver.html](#)], `JdbcDatabaseMetaData` [[../src/org/hsqldb/jdbc/jdbcDatabaseMetaData.html](#)], `JdbcResultSet` [[../src/org/hsqldb/jdbc/jdbcResultSet.html](#)], `JdbcStatement` [[../src/org/hsqldb/jdbc/jdbcStatement.html](#)], and `JdbcPreparedStatement` [[../src/org/hsqldb/jdbc/jdbcPreparedStatement.html](#)] list all the supported JDBC methods together with information that is specific to HSQLDB. JDBC methods are broadly divided into: connection related methods, metadata methods and database access methods. The database access methods use SQL commands to perform actions on the database and return the results either as a Java primitive type or as an instance of the `java.sql.ResultSet` class.

You can use Database Manager or other Java database access tools to explore your database and update it with SQL commands. These programs use JDBC internally to submit your commands to the database engine and to display the results in a human readable format.

The SQL dialect used in HSQLDB is as close to the SQL92 and SQL200n standards as it has been possible to achieve so far in a small-footprint database engine. The full list of SQL commands is in the SQL Syntax chapter.

## Different Types of Tables

HSQLDB supports TEMP tables and three types of persistent tables.

TEMP tables are not written to disk and last only for the lifetime of the Connection object. Each TEMP table is visible only from the Connection that was used to create it; other concurrent connections to the database will not have access to the table.

The three types of persistent tables 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 `<dbname>.script` file. The script file is read the next time the database is opened, and the MEMORY tables are recreated with all their contents. So unlike TEMP table, the default, MEMORY tables are persistent.

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 are new to version 1.7.0 and 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. Two commands are needed to set up a TEXT table as detailed in the Text Tables chapter.

With memory-only databases (see above), both MEMORY table and CACHED table declarations are treated as declarations for non-persistent memory tables. TEXT table declarations are not allowed in this mode.



## Constraints and Indexes

HSQldb supports PRIMARY KEY, NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints. In addition, it supports UNIQUE or ordinary indexes. This support is fairly comprehensive and covers multi-column constraints and indexes, plus cascading updates and deletes for foreign keys.

HSQldb 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. Because of this, you should not create duplicate user-defined indexes on the same column sets covered by these constraints. This would result in unnecessary memory and speed overheads. See the discussion in the SQL Issues chapter for more information.

Indexes are crucial for adequate query speed. When queries joining multiple tables are used, there must be an index on each joined column of each table. When range or equality conditions are used e.g. `SELECT ... WHERE a >10 AND b = 0`, an index is required on column used in the condition. Indexes have no effect on ORDER BY clauses or some LIKE conditions.

As a rule of thumb, HSQldb is capable of processing queries and returning over 100,000 rows per second. Any query that runs into several seconds should be checked and indexes should be added to the relevant columns of the tables if necessary.

## SQL Support

The SQL syntax supported by HSQldb is essentially that specified by the SQL Standard (92 and 200n). Not all the features of the Standard are supported and there are some proprietary extensions. In 1.7.2 the behaviour of the engine is far more compliant with the Standards than with older versions. The main changes are

- correct treatment of NULL column values in joins, in UNIQUE constraints and in query conditions
- correct processing of selects with JOIN and LEFT OUTER JOIN
- correct processing of aggregate functions contained in expressions or containing expression arguments

The supported commands are listed in the SQL Syntax chapter. For a well written basic guide to SQL you can consult PostgreSQL: Introduction and Concepts [<http://www.postgresql.org/docs/awbook.html>] by Bruce Momjian, which is available on the web. Most of the SQL coverage in the book applies also to HSQldb. There are some differences in keywords supported by one and not the other engine (ALL, ANY, OUTER, OID's, etc.) or used differently (IDENTITY/SERIAL, LIMIT, TRIGGER, SEQUENCE, etc.).

## JDBC Support

In 1.7.2, support for JDBC2 has been significantly extended and some features of JDBC3 are also supported. The relevant classes are thoroughly documented. See the JavaDoc for `org.hsqldb.jdbcXXXX` [`./src/`] classes.

---

# Chapter 2. SQL Issues

Fred Toussi, HSQLDB Development Group <ft@cluedup.com>

Copyright 2002-2004 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 HSQLDB Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.

\$Date: 2004/08/09 17:10:11 \$

## Purpose

Many questions repeatedly asked in Forums and mailing lists are answered in this guide. If you want to use HSQLDB with your application, you should read this guide.

## SQL Standard Support

HSQLDB 1.7.3 supports the dialect of SQL defined by SQL standards 92, 99 and 2003. This means where a feature of the standard is supported, e.g. left outer join, the syntax is that specified by the standard text. Many features of SQL92 and 99 up to Advanced Level are supported and there is support for most of SQL 2003 Foundation and several optional features of this standard. However, certain features of the Standards are not supported so no claim is made for full support of any level of the standards.

The SQL Syntax chapter of this guide SQL Syntax lists all the keywords and syntax that is supported. When writing or converting existing SQL DDL (Data Definition Language) and DML (Data Manipulation Language) statements for HSQLDB, you should consult the supported syntax and modify the statements accordingly.

Several words are reserved by the standard and cannot 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. HSQLDB does not currently prevent you from using a reserved word if it does not support its use or can distinguish it. For example ANY and BEGIN are reserved words that are not currently supported by HSQLDB and are allowed as names of tables or columns. You should avoid the use of such words as future versions of HSQLDB are likely to support the words and will reject your table definitions or queries. The full list of SQL reserved words is in the source of the org.hsqldb.Token class.

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

## Constraints and Indexes

### Primary Key Constraints

Before 1.7.0, a CONSTRAINT <name> PRIMARY KEY was translated internally to a unique index and, in addition, a hidden column was added to the table with an extra unique index. From 1.7.0 both single-column and multi-column PRIMARY KEY constraints are supported. They are supported by a unique index on the primary key column(s) specified and no extra hidden column is maintained for these indexes.

### Unique Constraints

According to the SQL standards, a unique constraint on a single column means no two values are equal unless one of them is NULL. This means you can have one or more rows where the column value is NULL.

A unique constraint on multiple columns (c1, c2, c3, ..) means that no two sets of values for the columns are equal unless at least one of them is NULL. Each single column taken by itself can have repeat values. The following example satisfies a UNIQUE constraint on the two columns:

### Example 2.1. 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

Since version 1.7.2 the behaviour of UNIQUE constraints and indexes with respect to NULL values has changed to conform to SQL standards. A row, in which the value for any of the UNIQUE constraint columns is NULL, can always be added to the table. So multiple rows can contain the same values for the UNIQUE columns if one of the values is NULL.

## Unique Indexes

In 1.7.3, user defined UNIQUE indexes can still be declared but they are deprecated. You should use a UNIQUE constraint instead.

CONSTRAINT <name> UNIQUE always creates internally a unique index on the columns, as with previous versions, so it has exactly the same effect as the deprecated UNIQUE index declaration.

## FOREIGN KEYS

From version 1.7.0, HSQLDB features single and multiple column foreign keys. A foreign key can also be specified to reference a target table without naming the target column(s). In this case the primary key column(s) of the target table is used as the referenced column(s). Each pair of referencing and referenced columns in any foreign key should be of identical type. When a foreign key is declared, a unique constraint (or primary key) must exist on the referenced columns in the primary key table. A non-unique index is automatically created on the referencing columns. For example:

```
CREATE TABLE child(c1 INTEGER, c2 VARCHAR, FOREIGN KEY (c1, c2) REFERENCES par
```

There must be a UNIQUE constraint on columns (p1, p2) in the table named "parent". A non-unique index is automatically created on columns (c1, c2) in the table named "child". Columns p1 and c1 must be of the same type (INTEGER). Columns p2 and c2 must be of the same type (VARCHAR).

## Indexes and Query Speed

HSQLDB does not use indexes to improve sorting of query results. But indexes have a crucial role in improving query speed. If no index is used in a query on a single table, such as a DELETE query, then all the rows of the table must be examined. With an index on one of the columns that is in the WHERE

clause, it is often possible to start directly from the first candidate row and reduce the number of rows that are 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 t1.c3, only about 4 rows are checked to get the first result row.

Indexes are automatically created for primary key and unique columns. Otherwise you should define an index using the `CREATE INDEX` command.

Note that in HSQLDB a unique 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 1.7.3, a multi-column index will speed up queries that contain joins or values on ALL the columns. 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 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.

As a result of the improvements to multiple key indexes, the order of declared columns of the index or constraint has less affect on the speed of searches than before. If the column that contains more diverse values appears first, the searches will be slightly faster.

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.

Query speed depends a lot 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.

1.7.3 features automatic, on-the-fly indexes for views and subselects that are used in a query. An index is added to a view when it is joined to a table or another view.

## Where Condition or Join

Using WHERE conditions to join tables is likely to reduce execution speed. For example the following

---

query will generally be slow, even with indexes:

```
SELECT ... FROM TA, TB, TC WHERE TC.COL3 = TA.COL1 AND TC.COL3=TB.COL2 AND TC.
```

The query implies `TA.COL1 = TB.COL2` but does not explicitly set this condition. If TA and TB each contain 100 rows, 10000 combinations will be joined with TC to apply the column conditions, even though there may be indexes on the joined columns. With the JOIN keyword, the `TA.COL1 = TB.COL2` condition has to be explicit and will narrow down the combination of TA and TB rows before they are joined with TC, resulting in much faster execution with larger tables:

```
SELECT ... FROM TA JOIN TB ON TA.COL1 = TB.COL2 JOIN TC ON TB.COL2 = TC.COL3 W
```

The query can be speeded up a lot more if the order of tables in joins are changed, so that `TC.COL1 = 1` is applied first and a smaller set of rows are joined together:

```
SELECT ... FROM TC JOIN TB ON TC.COL3 = TB.COL2 JOIN TA ON TC.COL3 = TA.COL1 W
```

In the above example the engine automatically applies `TC.COL4 = 1` to TC and joins only the set of rows that satisfy this condition with other tables. Indexes on `TC.COL4`, `TB.COL2` and `TA.COL1` will be used if present and will speed up the query.

## Subqueries and Joins

Using joins and setting up the order of tables for maximum performance applies to all areas. For example, the second query below should generally be much faster if there are indexes on `TA.COL1` and `TB.COL3`:

### Example 2.2. Query comparison

```
SELECT ... FROM TA WHERE TA.COL1 = (SELECT MAX(TB.COL2) FROM TB WHERE TB.COL3  
SELECT ... FROM (SELECT MAX(TB.COL2) C1 FROM TB WHERE TB.COL3 = 4) T2 JOIN TA
```

The second query turns `MAX(TB.COL2)` into a single row table then joins it with TA. With an index on `TA.COL1`, this will be very fast. The first query will test each row in TA and evaluate `MAX(TB.COL2)` again and again.

## Types and Arithmetic Operations

Table columns of all types supported by HSQLDB can be indexed and can feature in comparisons. Types can be explicitly converted using the `CONVERT()` library function, but in most cases they are converted automatically. It is recommended not to use indexes on `LONGVARIABLE`, `LONGVARIABLE` and `OTHER` columns, as these indexes will probably not be allowed in future versions.

Previous versions of HSQLDB featured poor handling of arithmetic operations. For example, it was not possible to insert `10/2.5` into any `DOUBLE` or `DECIMAL` column. In 1.7.0, full operations are possible with the following rules:

TINYINT, SMALLINT, INTEGER, BIGINT, NUMBER and DECIMAL (without a decimal point) are supported integral types and map to byte, short, int, long and BigDecimal in Java. 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, although the actual Java type used for handling TINYINT is `java.lang.Integer`.

REAL, FLOAT, DOUBLE are all mapped to double in Java.

DECIMAL and NUMERIC are mapped to `java.math.BigDecimal` and can have very large numbers of digits before or after the decimal point.

## Integral Types

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

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;
```

would return a result set where the type of the first column is `java.lang.Integer` and the second column is `java.lang.Long`. However,

```
SELECT MAX(a) + 0, MAX(b) + 0 FROM t;
```

would return `java.lang.Long` and `BigDecimal` values, generated as a result of uniform type promotion for all the return values.

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.

## Other Numeric Types

In SQL statements, numbers 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 `PreparedStatement.setDouble()` or `setFloat()` is used, the value is treated as a DOUBLE automatically.

When a REAL, FLOAT or DOUBLE (all synonymous) is part of an expression, the type of the result is DOUBLE.

Otherwise, when no `DOUBLE` value exists, if a `DECIMAL` or `NUMBER` value is part an expression, the type of the result is `DECIMAL`. The result can be retrieved from a `ResultSet` in the required type so long as it can be represented. This means `DECIMAL` values can be converted to `DOUBLE` unless they are beyond the `Double.MIN_VALUE - Double.MAX_VALUE` range. Similar to integral values, when the result of an expression is stored in a table column, it has to fit in the target column, otherwise an error is returned.

The distinction between `DOUBLE` and `DECIMAL` is important when a division takes place. When the terms are `DECIMAL`, 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. With a `DOUBLE` term, the scale will reflect the actual result of the operation. 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; the resulting scale is the sum of the scales of the two terms when multiplication is performed.

`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.

## Bit and Boolean Types

Since 1.7.2, `BIT` is simply an alias for `BOOLEAN`. The primary representation of `BOOLEAN` column is `'true'` or `'false'` either as the boolean type or as strings when used from `JDBC`. This type of column can also be initialised using values of any numeric type. In this case `0` is translated to `false` and any other value such as `1` is translated to `true`.

Since 1.7.3 the `BOOLEAN` type conforms to the SQL standards and supports the `UNDEFINED` state in addition to `TRUE` or `FALSE`. `NULL` values are treated as undefined. This improvement affects queries that contain `NOT IN`. See the test text file, `TestSelfNot.txt`, for examples of the queries.

## Storage and Handling of Java Objects

Since version 1.7.2 this support has improved and 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`, or `WHERE object1 = ?` 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 return errors when normal column values are 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 ...`) but this is highly likely to be disallowed in future. So please use columns of type `OTHER` only to store your objects and nothing else.

## Type Size, Precision and Scale

Prior to 1.7.2, all table column type definitions with a column size, precision or scale qualifier were accepted and ignored. Since version 1.7.2, such qualifiers must conform to the SQL standards. For example `INTEGER(8)` is no longer acceptable.

Since 1.7.2, the qualifiers are still ignored unless you set a database property. `SET PROPERTY`

"`sql.enforce_strict_size`" `TRUE` will enforce sizes for `CHARACTER` or `VARCHAR` columns and pad any strings when inserting or updating a `CHARACTER` column. The qualifiers are still ignored for `DECIMAL` and `DOUBLE` numeric types.

Please note that casting a value to a qualified `CHARACTER` type will not result in truncation or padding as you might expect. So you cannot rely on a test such as `CAST (mycol AS VARCHAR(2)) = 'xy'` to find the values beginning with 'xy'. Use `SUBSTRING(mycol FROM 1 FOR 2)` instead.

## Sequences and Identity

The `SEQUENCE` keyword has been introduced in 1.7.2 with a subset of the SQL 200n standard syntax. Corresponding SQL 200n syntax for `IDENTITY` columns has also been introduced.

### Identity Auto-Increment Columns

Each table can contain one auto-increment column, known as the `IDENTITY` column. An `IDENTITY` column is always treated as the primary key for the table (as a result, multi-column primary keys are not possible with an `IDENTITY` column present). Support has been added for `CREATE TABLE <tablename>(<colname> IDENTITY, ...)` as a shortcut.

In 1.7.2 the SQL standard syntax is used by default, which allows the initial value to be specified. The supported form is `(<colname> INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH n, [INCREMENT BY m])PRIMARY KEY, ...)`. Support has also been added for `BIGINT` identity columns. As a result, an `IDENTITY` column is simply an `INTEGER` or `BIGINT` column with its default value generated by a sequence generator.

When you add a new row to such a table using an `INSERT INTO <tablename> ...;` statement, you can use the `NULL` value 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 connection. 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 on the database.

## Sequences

The SQL 200n 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. 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 2.3. Numbering returned rows of a `SELECT` in sequential order

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

Please note that the semantics of sequences is not exactly the same as defined by SQL 200n. For example if you use the same sequence twice in the same row insert query, you will get two different values, not the same value as required by the standard.

You can query the `SYSTEM_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.

## Issues with Transactions

HSQldb supports transactions at the `READ_UNCOMMITTED` level, also known as level 0 transaction isolation. This means that during the lifetime of a transaction, other connections to the database can see the changes made to the data. Transaction support works well in general. Reported bugs concerning transactions being committed if the database is abruptly closed have been fixed. However, the following issues may be encountered only with multiple connections to a database using transactions:

If two transactions modify the same row, no exception is raised when both transactions are committed. This can be avoided by designing your database in such a way that application data consistency does not depend on exclusive modification of data by one transaction.

When an `ALTER TABLE .. INSERT COLUMN` or `DROP COLUMN` command results in changes to the table structure, the current session is committed. If an uncommitted transaction started by another connections has changed the data in the affected table, it may not be possible to roll it back after the `ALTER TABLE` command. This may also apply to `ADD INDEX` or `ADD CONSTRAINT` commands. It is recommended to use these `ALTER` commands only when it is known that other connections are not using transactions.

After a `CHECKPOINT` command is issued, uncommitted transactions can be continued, committed, or rolled back. However, if the database is not subsequently closed properly with the `SHUTDOWN` command, any such transaction that still remains uncommitted at the time of shutdown, is part committed (to the state at `CHECKPOINT`) at the next startup. It is recommended not to use the `CHECKPOINT` command when there are any uncommitted transactions.

## New Features and Changes

In 1.7.2 many enhancements were made for better SQL support. These are listed in the SQL Syntax chapter and in `../changelog_1_7_2.txt`. Functions and expressions such as `POSITION()`, `SUBSTRING()`, `NULLIF()`, `COALESCE()`, `CASE ... WHEN .. ELSE` etc. are among them. Other enhancements may not be very obvious in the documentation but can result in changes of behaviour from previous versions. Most significant among these are handling of `NULL` values in joins (null columns are no longer joined) and `OUTER` joins (the results are now correct). You should test your applications with the new version to ensure they do not rely on past incorrect behaviour of the engine. The engine will evolve in future versions towards full SQL standard support, so it is best not to rely on any non-standard feature of the current version.

---

# Chapter 3. UNIX Quick Start

## *How to quickly get Hsqldb up and running on UNIX, including Mac OS X*

Blaine Simpson, HSQLDB Development Group  
<blaine.simpson@admc.com>

\$Date: 2004/07/16 13:13:07 \$

## Purpose

This chapter explains how to quickly install, run, and use HSQLDB version 1.7.2 on UNIX.

HSQLDB has lots of great optional features. I intend to cover very few of them. I do intend to cover what I think is the most common UNIX setup: To run a multi-user database with permanent data persistence. (By the latter I mean that data is stored to disk so that the data will persist across database shutdowns and startups). I also cover how to run Hsqldb as a system daemon.

## Installation

Go to <http://sourceforge.net/projects/hsqldb> and click on the "files" link. Look for "hsqldb\_1\_7\_2" under lower-case "hsqldb". Click on "show only this release" link right after "hsqldb\_1\_7\_2".

Click the "hsqldb\_1\_7\_2" link to find out what version of Java this binary HSQLDB distribution was built with. Don't hold me to it, but if you get your distro from us at SourceForge, it will probably be built with Java 1.4. Choose a binary package format that will work with your UNIX variant and which supports your Java version. Otherwise choose the hsqldb\_1\_7\_2.zip file. Click the filename to download it. (If you have an older version of Java and there's nothing preventing you from upgrading it, you'll probably be happier in the end if you upgrade Java rather than downgrading HSQLDB).

If you want an rpm, then click "hsqldb" in the "free section" of <http://www.jpackage.org/>. Hopefully, the JPackage folk will document what JVM versions their rpm will support (currently they document this neither on their site nor within the package itself). Download the package you want, making sure that you get version 1.7.2 of HSQLDB. (I really can't document how to download from a site that is totally beyond my control).

### Note

It could very well happen that some of the file formats which I discuss here 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 filename.pkg
```

Installing from a .rpm file

This is a Linux rpm package. 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.

Installing from a .zip file

Extract the zip file to the parent directory of the new **HSQLDB** home. You don't need to create the **HSQLDB\_HOME** directory because the extraction will create it for you with the right name)

```
cd parent/of/new/hsqldb/home
unzip /path/to/file.zip
```

All the files in the zip archive will be extracted to underneath a new `hsqldb` directory.

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 `hsqldb` system is `hsqldb.jar`, which resides in the directory `lib`.

## 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`.

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 read `HSQLDB_HOME/readme.txt` and `HSQLDB_HOME/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 3.1. 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 jar
```

This will build a new `HSQLDB_HOME/lib/hsqldb.jar`.

See the Building Hsqldb version 1.7.2 appendix if you want to build anything other than `hsqldb.jar` with all default settings.

## Setting up a Hsqldb Persistent Database Instance and a Hsqldb Server

If you installed from an OS-specific package, you may already have a database instance and server 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 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 instance files and 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 -m -c 'HSQLDB Database Owner' -s /bin/ksh -m hsqldb
```

(BSD-variant users can use a similar `pw useradd hsqldb...` command).

2. Become the **HSQLDB\_OWNER**. Copy the sample file **HSQLDB\_HOME/src/org/hsqldb/sample/sample-server.properties** to the **HSQLDB\_OWNER**'s home directory and rename it to `server.properties`.

```
# Hsqldb Server cfg file.
# See the Advanced Topics chapter of the Hsqldb User Guide.
```

```
server.database.0 file:db0/db0
```

Since the value of the first database (`server.database.0`) begins with `file:`, the database instance will be persisted to a set of files in the specified directory with names beginning with the specified name. You can read about how to specify other database instances of various types, and how to make settings for the listen port and many other things, in the Advanced Topics chapter.

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

```
java org.hsqldb.Server &
```

This will start the Server process in the background, and will create your new database instance "db0". Continue on when you see the message containing "HSQLDB server... is online".

## Accessing your Database

Copy the file **HSQLDB\_HOME/src/org/hsqldb/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,v 1.11 2004/07/18 21:34:35 unsaved Exp $
# This is a sample SqlTool configuration file, a.k.a. rc file.
# You can run SqlTool right now by copying this file to your home directory
# and running
# java -jar /path/to/hsqldb.jar mem
# This will access the first urlid definition below in order to use a
# personal Memory-Only database.
# If you have the least concerns about security, then secure access to
```

```

# your sqltool.rc file.
# See the documentation for SqlTool for various ways to use this file.

# A personal Memory-Only database.
urlid mem
url jdbc:hsqldb:mem:memdbid
username sa
password

# 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:hsq://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 ../hsqsqltool.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 urlid for a Postgresql database.
# You will need to put the org.postgresql.Driver class into your
# classpath.
# The postgresql jar will be named postgresql.jar (if you built Postgresql
# from source), or something like pg73bljdbc3.jar or jdbc7.2x-1.2.jar.
# You can obtain it from a client or server Postgresql installation, or
# download it from http://jdbc.postgresql.org/download.html.
# Notice that the jar file names (other than "postgresql.jar") contain both
# the target Postgresql server version and the client-side JDBC level (which
# is determined by your client-side Java version, as explained at
# http://jdbc.postgresql.org/download.html).
# I recommend the latest production version for your JDBC version. The
# later JDBC drivers work better even with older Postgresql servers.
# (E.g. \dt won't list owners with an older driver).
# N.b.: Suse Linux 9.1 users should download a new driver from the PG site,
# since Suse distributes the 7.3 drivers with Postgresql 7.4 (why???).
# 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 ../hsqsqltool.jar...".
# Put both the HSQLDB jar and the Postgresql jar in your classpath (and
# export!) and run something like "java org.hsqldb.util.SqlTool...".
# N.b.: I notice that Postgresql is unusual in that it does not do an
# implicit commit before DDL commands. If you get an error message
# "... cannot run inside a transaction block", just run "commit;" and retry.

```

```
#urlid commerce
#url jdbc:postgresql://dbsvr2/commercedb
#username blaine
#password obscured
#driver org.postgresql.Driver
#####

#####
# Template for a TLS-encrypted HSQLDB Server.
# Remember that the hostname in hsqs (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:hsqs://db.admc.com:9001/lm2
#username blaine
#password asecret
#truststore /home/blaine/ca/db/db-trust.store
#####
```

We will be using the "localhost-sa" sample urlid definition from the config file. The JDBC URL for this urlid is `jdbc:hsqldb:hsq://localhost`. That is the URL for the default database instance of a HSQLDB Server running on the default port of the local host. You can read about URLs to connect to other instances and other servers in the Advanced Topics chapter.

Run `SqlTool`.

```
java -jar path/to/hsqldb.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";
```

When you're finished playing, exit with the command `\q`.

If you changed the SA password, then you need to fix the password in the `sqltool.rc` file accordingly.

You can, of course, also access the database with any JDBC client program. See the First JDBC Client Example appendix. 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.DatabasesManagerSwing`, 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 access to the `hsqldb.jar` file and to an `sqltool.rc` file. See the `SqlTool` chapter 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 database instance. HSQLDB accounts are database-instance-specific, not `Server`-specific.

There are two classes of database accounts, Admin accounts and non-Admin accounts. Admins have privileges to do anything, non-Admins may be granted some privileges, but may never create or own database objects. When you first create a hsqldb database, it has only one database user-- SA, an Admin account, with no password set. You should set a password (as described above). You can create as many additional Admin users as you wish. Each Admin user has a unique user name (and optional password), but these accounts are otherwise indistinguishable. These accounts are created by appending the keyword "ADMIN" to the `CREATE USER` command.

If you create a user without the ADMIN tag, it will be a Non-Admin account. These users can not create or own objects, and, by default, they can't use any database objects. The user will then be able to perform operations which have been granted to the pseudo-user PUBLIC. To give the user additional privileges (even the privilege to read data), an Admin user must grant those rights to the user (or to PUBLIC).

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 any object that needs to be accessed by multiple database users (with the significant exception of any data which you want to keep secret).

## Shutdown

Do a clean database shutdown when you are finished with the database instance. You need to connect up as SA or some other Admin user, of course. With `SqlTool`, you can run

```
java -jar path/to/hsqldb.jar --noinput --sql 'shutdown;' localhost-sa
```

You don't have to worry about stopping the `Server` because it shuts down automatically when all served database instances 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 `Server` will run as root-- it usually should not).

The main purpose of the `init` script is to start up a `Server` with the database instances specified in your `server.properties` file; and to shut down all of those instances *plus* additional `urlids` which you may (optionally) list in your `init` script config file. These `urlids` must all have entries in a `sqltool.rc` 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.

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 HP-UX; 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 its 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

1. Copy the init script `hsqldb` from **HSQLDB\_HOME**/bin into 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. Look at the init script and see what the value of `CFGFILE` is for your UNIX platform. You need to copy the sample config file **HSQLDB\_HOME**/src/org/hsqldb/sample/sample-hsqldb.cfg to that location. Edit the config file according to the instructions in it.

```
# $Id: sample-hsqldb.cfg,v 1.9 2004/07/15 18:54:47 unsaved Exp $

# Sample configuration file for HSQLDB database server.
# See the "UNIX Quick Start" chapter of the Hsqldb 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.

# Thanks to Meikel Bisping for his contributions. -- Blaine

JAVA_EXECUTABLE=/usr/java/j2sdk1.4.2_02/bin/java

# Unless you copied a hsqldb.jar file from another system, this typically
# resides at $HSQLDB_HOME/lib/hsqldb.jar, where $HSQLDB_HOME is your HSQLDB
# software base directory.
HSQLDB_JAR_PATH=/home/blaine/hsqldb-dev/lib/hsqldb.jar

# Where the file "server.properties" (or "webserver.properties") resides.
SERVER_HOME=/home/blaine/db

# What UNIX user the Server/WebServer process 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.
# You do need to run as root if your Server/WebServer will run on a privileged
# (< 1024) port.
# If you really do want to run as root, comment out the HSQLDB_OWNER setting
# completely. I.e., do not set it to root. This will run Server/Webserver
# without any "su" at all.
HSQLDB_OWNER=blaine
```

```

# 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 if you are running lots of DB instances or have a slow server.
#MAX_START_SECS=200
# Ditto for this one
#SU_ECHO_SECS=1

# Time to allow for JVM to die after all HSQLDB instances stopped.
# Defaults to 1.
#MAX_TERMINATE_SECS=0

# These are "urlid" values from a SqlTool authentication file
# ** IN ADDITION TO THOSE IN YOUR server.properties OR webserver.properties **
# file. All server.urlid.X values from your properties file will automatically
# be started/stopped/tested. $SHUTDOWN_URLIDS is for additional urlids which
# will be stopped. (Therefore, most users will not set this at all).
# Separate multiple values with white space. NO OTHER SPECIAL CHARACTERS!
# Make sure to quote the entire value if it contains white space separator(s).
# Defaults to none (i.e., only urlids set in properties file will be stopped).
#SHUTDOWN_URLIDS='sa mygms'

# 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 HSQLDB User Guide if you don't understand
# this).
#AUTH_FILE=/home/blaine/sqltool.rc

# Set to 'WebServer' to start a HSQLDB WebServer instead of a Server.
# Defaults to 'Server'.
#TARGET_CLASS=WebServer

# Server-side classpath IN ADDITION TO the HSQLDB_JAR_PATH set above.
# The classpath here is *earlier* than HSQLDB_JAR_PATH, to allow you
# override classes in the HSQLDB_JAR_PATH jar file.
# In particular, you will want to add classpath elements to give access of
# all of your store procedures (store procedures are documented in the
# HSQLDB User Guide in the SQL Syntax chapter.
# SERVER_ADDL_CLASSPATH=/home/blaine/storedprocs.jar:/usr/dev/dbutil/classes

# For TLS encryption for your Server, set these two variables.
# N.b.: If you set these, then make this file unreadable to non-root users!!!
# See the TLS chapter of the HSQLDB 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).
#TLS_KEYSTORE=/path/to/jks/server.store
#TLS_PASSWORD=password

# Any JVM args for the invocation of the JDBC client used to verify DB
# instances and to shut them down (SqlToolSprayer).
# For multiple args, put quotes around entire value.
#CLIENT_JVMARGS=-Djavax.net.debug=ssl

# Any JVM args for the server.
# For multiple args, put quotes around entire value.
#SERVER_JVMARGS=-Xmx512m

```

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 (for stops) 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.
4. Edit your `server.properties` file. For every `server.database.X` that you have defined, set a property of name `server.urlid.X` to the urlid for an Administrative user for that database instance.

### Example 3.1. server.properties fragment

```
server.database.0=file://home/hsqldb/data/db1
server.urlid.0=localhostdb1
```

### Warning

Make sure to add a urlid for each and every database instance. If you don't then the init script will never know about databases that become inaccessible and will give false diagnostics.

For this example, you would need to define the urlid `localhostdb1` in your `sqltool.rc` file.

### Example 3.2. example sqltool.rc stanza

```
urlid localhostdb1
url jdbc:hsqldb:hsq1://localhost
username sa
password secret
```

5. **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 Server** 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.

6. 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.

## Troubleshooting the Init Script

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 the `urlid(s)` listed in the `server.properties` or `web-server.properties` are correct, and verify that you can run `SqlTool` as root to connect to the instances. (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. If these 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 or email me at [blaine.simpson@admc.com](mailto:blaine.simpson@admc.com) [mailto:blaine.simpson@admc.com?Subject=hsqldb-unix]. If you email me, 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
```

---

# Chapter 4. Advanced Topics

Fred Toussi, HSQLDB Development Group <ft@cluedup.com>

Copyright 2002-2004 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 HSQLDB Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.  
\$Date: 2004/11/24 23:06:35 \$

## Purpose

Many questions repeatedly asked in Forums and mailing lists are answered in this guide. If you want to use HSQLDB with your application, you should read this guide. This document covers system related issues. For issues related to SQL see the SQL Issues chapter.

## Connections

The normal method of accessing an HSQLDB database is via the JDBC Connection interface. An introduction to different methods of providing database services and accessing them can be found in the SQL Issues chapter. Details and examples of how to connect via JDBC are provided in our JavaDoc for `JdbcConnection` [[./src/org/hsqldb/jdbc/jdbcConnection.html](http://hsqldb.org/src/org/hsqldb/jdbc/jdbcConnection.html)].

Version 1.7.2 introduces a uniform method of distinguishing between different types of connection, alongside new capabilities to provide access to multiple databases. The common driver identifier is `jdbc:hsqldb:` followed by a protocol identifier (`mem:` `file:` `res:` `hsqldb:` `http:` `hsqldb:` `https:`) then followed by host and port identifiers in the case of servers, then followed by database identifier.

**Table 4.1. Hsqldb URL Components**

Driver and Protocol	Host and Port	Database
<code>jdbc:hsqldb:mem:</code>	not available	<code>accounts</code>
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, <code>jdbc:hsqldb:.</code> creates or connects to the same database as the new form for the URL, <code>jdbc:hsqldb:mem:.</code>		
<code>jdbc:hsqldb:file:</code>	not available	<code>mydb</code> <code>/opt/db/accounts</code> <code>C:/data/mydb</code>
The file path specifies the database file. In the above examples the first one refers to a set of <code>mydb.*</code> files in the directory where the <code>javac</code> command for running the application was issued. The second and third examples refer to absolute paths on the host machine.		
<code>jdbc:hsqldb:res:</code>	not available	<code>/adirectory/dbname</code>
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 <code>/adirectory</code> above stands for a directory in		

Driver and Protocol	Host and Port	Database
one of the jars.		
jdbc:hsqldb:hsq1: jdbc:hsqldb:hsq1s: jdbc:hsqldb:http: jdbc:hsqldb:https:	//localhost //192.0.0.10:9500 / /dbserv- er.somedomain.com	/an_alias /enrollments /quickdb
<p>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 <code>server.properties</code> file to 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 <code>server.properties</code> or <code>web-server.properties</code> define the database aliases listed above and accessible to clients to refer to different file and in-memory databases.</p> <pre> database.0=file:/opt/db/accounts dbname.0=an_alias  database.1=file:/opt/db/mydb dbname.1=enrollments  database.2=mem:adatabase dbname.2=quickdb </pre> <p>The old form for the server URL, e.g., <code>jdbc:hsqldb:hsq1//localhost</code> connects to the same database as the new form for the URL, <code>jdbc:hsqldb:hsq1//localhost/</code> where the alias is a zero length string. In the example below, the database files <code>lists.*</code> in the <code>/home/dbmaster/</code> directory are associated with the empty alias:</p> <pre> database.3=/home/dbmaster/lists dbname.3= </pre>		

## Connection properties

Each new JDBC Connection to a database can specify connection properties. The properties `user` and `password` are always required. In 1.7.2 the following optional properties can also be used.

Connection properties are specified either by establishing the connection via the:

```
DriverManager.getConnection (String url, Properties info);
```

method call, or the property can be appended to the full Connection URL.

**Table 4.2. Connection Properties**

<code>get_column_name</code>	<code>true</code>	column name in ResultSet
<p>This property is used for compatibility with other JDBC driver implementations. When <code>true</code> (the default), <code>ResultSet.getColumnName(int c)</code> returns the underlying column name</p>		

When false, the above method returns the same value as `ResultSet.getColumnLabel(int column)` Example below:

```
jdbc:hsqldb:hsq://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 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
```

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 is used to specify properties to enforce more strict SQL adherence, or to change `cache_scale` or similar properties before the database files are created.

## Properties Files

HSQldb relies on a set of properties files for different settings. Version 1.7.0 streamlines property naming and introduces a number of new properties (in this document, all references to versions 1.7.0 also apply to versions 1.7.1 and 1.7.2 unless stated otherwise). This process will continue with future versions and the properties will be used in a hierarchical manner.

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).

The properties files and the settings stored in them are as follows:

**Table 4.3. Hsqldb Server Properties Files**

File Name	Location	Function
<code>server.properties</code>	the directory where the command to run the <code>Server</code> class is issued	settings for running HSQldb as a database server communicating with the HSQL protocol
<code>webserver.properties</code>	the directory where the command to run the <code>WebServer</code> class is issued	settings for running HSQldb as a database server communicating with the HTTP protocol
<code>&lt;dbname&gt;.properties</code>	the directory where all the files for a database are located	settings for each particular database

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 properties file for each database is generated by the database engine. This file can be edited after closing the database. In 1.7.2, some property values can be changed via SQL commands.

## Server and Web Server Properties

In both `server.properties` and `webserver.properties` files, supported values and their defaults are as follows:

**Table 4.4. Property File Properties**

Value	Default	Description
<code>server.database.0</code>	<code>test</code>	the path and file name of the first database file to use
<code>server.dbname.0</code>	<code>" "</code>	lowercase server alias for the first database file
<code>server.urlid.0</code>	<code>NONE</code>	SqlTool urlid used by UNIX init script. (This property is not used if you are running Server/Webserver on a platform other than UNIX, or if you are not using our UNIX init script).
<code>server.silent</code>	<code>true</code>	no extensive messages displayed on console
<code>server.trace</code>	<code>false</code>	JDBC trace messages displayed on console

In 1.7.2, each server can serve up to 10 different 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.{0-9}` property can use the `mem:`, `file:` or `res:` prefixes and properties as discussed above under CONNECTIONS. For example,

```
database.0=mem:temp;sql.enforce_strict_size=true;
```

Values specific to `server.properties` are:

**Table 4.5. Server Property File Properties**

Value	Default	Description
<code>server.port</code>	<code>9001</code>	TCP/IP port used for talking to clients. All databases are served on the same port.
<code>server.no_system_exit</code>	<code>true</code>	no <code>System.exit()</code> call when the database is closed

Values specific to `webserver.properties` are:



**Table 4.6. WebServer Property File Properties**

Value	Default	Description
server.port	80	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>	?	multiple entries such as .html=text/html define the mime types of the static files served by the web server. See the source for WebServer.java for a list.

All the above values can be specified on the command line to start the server by omitting the server. prefix.

## 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 in the form of a String and start the Server. An example of this can be found in the `org.hsqldb.test.TestBase` source.

### 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.

## Individual Database Properties

Each database has its own `<dbname>.properties` file as part of a small group of files which also includes `<dbname>.script` and `<dbname>.data`. The properties files contain key/value pairs for some important settings.

In version 1.7.2 a new SQL command allows some user-accessible database properties to be modified as follows:

```
SET PROPERTY "property_name" property_value
```

Properties that can be modified via `SET PROPERTY` are indicated in the table below. Other properties are indicated as `PROPERTIES FILE ONLY` and can be modified only by editing the `.properties` file after a shutdown and before a restart. The `*.properties` file can be edited after closing the database. Only the user-defined values listed below should ever be modified. Changing any other value will result in unexpected malfunction in database operations. Most of these values have been introduced for the new features since 1.7.0 and are listed below with their default values in different contexts:

**Table 4.7. Database-specific Property File Properties**

Value	Default	Description
readonly	no	whole database is read-only
<p>When true, the database cannot be modified in use. This setting can be changed to <code>yes</code> if the database is to be opened from a CD. Prior to changing this setting, the database should be closed with the <code>SHUTDOWN COMPACT</code> command to ensure consistency and compactness of the data. ( <code>PROPERTIES FILE ONLY</code> )</p>		
hsqldb.files_readonly	false	database files will not be written to
<p>When true, data in <code>MEMORY</code> tables can be modified and new <code>MEMORY</code> tables can be added. However, these changes are not saved when the database is shutdown. <code>CACHED</code> and <code>TEXT</code> tables are always read-only when this setting is true. ( <code>PROPERTIES FILE ONLY</code> )</p>		
hsqldb.first_identity	0	first identity value for a new table
<p>The first value assigned automatically to the identity column of new tables. ( <code>SET PROPERTY</code> )</p>		
sql.enforce_size	false	trimming and padding string columns
<p>When true, all <code>CHARACTER</code> and <code>VARCHAR</code> values that are in a row affected by an <code>INSERT INTO</code> or <code>UPDATE</code> statement are trimmed to the size specified in the SQL table definition. Also all char strings that are shorter than the specified size are padded with spaces. When false (default), stores the exact string that is inserted. ( <code>SET PROPERTY</code> )</p>		
sql.enforce_strict_size	false	size enforcement and padding string columns
<p>Conforms to SQL standards. When true, all <code>CHARACTER</code> and <code>VARCHAR</code> values that are in a row affected by an <code>INSERT INTO</code> or <code>UPDATE</code> statement are checked against the size specified in the SQL table definition. An exception is thrown if the value is too long. Also all char strings that are shorter than the specified size are padded with spaces. When false (default), stores the exact string that is inserted. ( <code>SET PROPERTY</code> )</p>		
sql.compare_in_locale	false	locale used for sorting
<p><code>CHARACTER</code> and <code>VARCHAR</code> columns are by default sorted according to POSIX standards. Setting the value to true will result in sorting in the character set of the current JRE locale.</p> <p>Changing this value for an existing database that contains cached tables will break the indexing and result in inconsistent operation. To avoid this, first change the value in the properties file, then open the database and issue the <code>SHUTDOWN COMPACT</code> command to recreate all the indexes.</p> <p>If set true, this setting affects all the database in the JVM. ( <code>PROPERTIES FILE ONLY</code> )</p>		
hsqldb.cache_scale	14	memory cache exponent
<p>Indicates the maximum number of rows of cached tables that are held in memory, calculated as <math>3 * (2^{**}value)</math> (three multiplied by (two to the power value)). The default results in up to <math>3 * 16384</math> rows from all cached tables being held in memory at any time.</p> <p>The value can range between 8-18. ( <code>SET PROPERTY</code> ). If the value is set via <code>SET PROPERTY</code> then it becomes effective after the next database <code>SHUTDOWN</code> or <code>CHECKPOINT</code>. ( <code>SET PROPERTY</code> )</p>		
hsqldb.cache_size_scale	10	memory cache exponent
<p>Indicates the average size of each row in the memory cache used with cached tables, calculated as <math>2^{**}value</math> (two to the power value). This result value is multiplied by the maximum number of rows</p>		

Value	Default	Description
		<p>defined by <code>hsqldb.cache_scale</code> to form the maximum number of bytes for all the rows in memory cache. The default results in 1024 bytes per row. This default, combined with the default number of rows, results in approximately 50MB of the <code>.data</code> file to be stored in the memory cache.</p> <p>The value can range between 6-20. ( <code>SET PROPERTY</code> ). If the value is set via <code>SET PROPERTY</code> then it becomes effective after the next database <code>SHUTDOWN</code> or <code>CHECKPOINT</code>. ( <code>SET PROPERTY</code> )</p>
<code>hsqldb.log_size</code>	200	size of log when checkpoint is performed
		<p>The value is the size in megabytes that the <code>.log</code> file can reach before an automatic checkpoint occurs. A checkpoint and rewrites the <code>.script</code> file and clears the <code>.log</code> file. The value can be changed via the <code>SET LOGSIZE nnn SQL</code> command.</p>
<code>runtime.gc_interval</code>	0	forced garbage collection
		<p>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.</p> <p>This should not be set when the database engine is acting as a server inside an exclusive JVM. The setting can be useful when the database is used in-process with the application with some Java Runtime Environments (JRE's). Some JRE's increase the size of the memory heap before doing any automatic garbage collection. This setting would prevent any unnecessary enlargement of the heap. Typical values for this setting would probably be between 10,000 to 100,000. ( <code>PROPERTIES FILE ONLY</code> )</p>
<code>hsqldb.nio_data_file</code>	<code>true</code>	use of nio access methods for the <code>.data</code> file
		<p>When HSQLDB is compiled and run in Java 1.4, setting this property to <code>false</code> will avoid the use of nio access methods, resulting in reduced speed. If the data file is larger than 512MB 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.</p> <p>( <code>SET PROPERTY</code> ). If used before defining any <code>CACHED</code> table, it applies to the current session, otherwise it comes to effect after a <code>SHUTDOWN</code> and restart or <code>CHECKPOINT</code>.</p>
<code>textdb.*</code>	0	default properties for new text tables
		<p>Properties that override the database engine defaults for newly created text tables. Settings in the text table <code>SET &lt;tablename&gt; SOURCE &lt;source string&gt;</code> command override both the engine defaults and the database properties defaults. Individual <code>textdb.*</code> properties are listed in the Text Tables chapter. ( <code>SET PROPERTY</code> )</p>

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.

### Note

Upgrading: From 1.7.0, the location of the database files can no longer be overridden by paths defined in the properties file. All files belonging to a database should reside in the same directory.

## JDBC Stream Based Methods

The JDBC interface has been significantly improved in 1.7.2. There is almost full support for JDBC2. Under JDK 1.4, there is also some support for JDBC3 methods. `ResultSetMetaData` methods are now fully supported, as are `ParameterMetaData` and `Savepoint` (JDBC3). When upgrading from previous versions, certain issues may arise as several JDBC methods that previously returned incorrect values or were not supported now return correct values. All changes have been documented in the Javadoc for the `jdbcXXX` classes.

Since 1.7.0, the `ResultSet` interface methods, `getAsciiStream()`, `getUnicodeStream()` and `getCharacterStream()` are supported to return byte or char values from `CHARACTER` columns and its variants. Complementary methods in `PreparedStatement`, `setAsciiStream()`, `setUnicodeStream()` and `setCharacterStream()` are also supported. Unlike most other databases, the `getString()` methods can be used to retrieve very long character strings and is faster than the stream based methods.

## 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. With other database engines, connection pools are used for reasons that may not apply to HSQLDB.

- To allow new queries to be performed while a time-consuming query is being performed in the background. This is not possible with HSQLDB as it blocks while performing the first query and deals with the next query once it has finished it.
- To limit the maximum number of simultaneous connections to the database for performance reasons. With HSQLDB this can be useful only if your application is designed in a way that opens and closes connections for each small task.
- To control transactions in a multi-threaded application. This can be useful with HSQLDB as well. For example, in a web application, a transaction may involve some processing between the queries or user action across web pages. A separate connection should be used for each session so that the work can be committed when completed or rolled back otherwise.

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 with versions prior to 1.7.2 the application program had to keep at least one connection to the database open, otherwise the database would have been closed and further attempts to create connections could fail. This is not necessary in 1.7.2, which does not automatically close an in-process database that is opened by establishing a connection. An explicit `SHTDOWN` command, with or without an argument, is required to close the database.

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 unsuccessful connection attempts when the application is under heavy load.

## Memory and Disk Use

Memory used by the program can be thought of as two distinct pools: memory used for table data, and memory used for building result sets and other internal operations. In addition, when transactions are

used, memory is utilised for storing the information needed for a rollback.

Since version 1.7.1, memory use has been significantly reduced compared to previous versions. 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 slots for int or reference variables (10 slots in previous versions). 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 slots for int or reference variables (12 slots in previous versions). As a result, a table with just one column of type `INTEGER` will have four objects per row, with a total of 10 slots 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.

The memory used for a result set row has fewer overheads (fewer slots and no index nodes) but still uses a lot of memory. All the rows in the result set are built in memory, so very large result sets may not be possible. In server mode databases, 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.

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 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 OFF`, lists of all insert, delete or update operations are stored in memory so that they can be undone when `ROLLBACK` is issued. Transactions that span hundreds of modification to data will take up a lot of memory until the next `COMMIT` or `ROLLBACK` clears the list.

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 `HSQldb`. For example, with Sun JVM version 1.3.0 the parameter `-Xmx256m` increases the amount to 256 MB.

1.7.2 uses a fast cache for immutable objects such as `Integer` or `String` 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.

## 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  $3 \times 16384$  rows. The `hsqldb.cache_scale` database 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 nearly 50,000 rows, including all the 40,000 larger rows.

In 1.7.2, an additional property, `hsqldb.cache_size_scale` is introduced. This property, combined with the `hsqldb.cache_scale` property, puts a limit in bytes on the total size of rows that are cached. When the default values is used for both properties, the limit on the total size of rows is approximately 50MB. (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_scale` or `hsqldb.cache_size_scale` database properties can be reduced. In the example above, if the `hsqldb.cache_size_scale` is reduced from 10 to 8, then the total binary size limit is reduced from 50MB to 12.5 MB. This will allow the number of cached rows to reach

50,000 small rows, but only 12,500 of the larger rows.

## Upgrading and Managing Databases

Any database not produced with the release version of HSQLDB 1.7.2 must be upgraded to this version. This includes databases created with the ALPHA versions of 1.7.2. The instructions under the Upgrading Using the SCRIPT Command section should be followed in all cases.

Once a database is upgraded to 1.7.2, it can no longer be used with Hypersonic or HSQLDB 1.6.x, 1.7.0 or 1.7.1.

There may be some potential problems in the upgrade which should be resolved by editing the .script file:

- Version 1.7.2 does not accept duplicate names for indexes.
- Version 1.7.2 does not accept duplicate names for table columns.
- Version 1.7.2 does not create the same type of index for foreign keys as previous versions.

## Upgrading Using the SCRIPT Command

In all versions of HSQLDB and Hypersonic 1.43, the `SCRIPT 'filename'` command (used as an SQL query) 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 1.7.2.

### Procedure 4.1. Upgrade Using SCRIPT procedure

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. You can now edit the `newversion.script` file to change any table and index definition so long as it is consistent with the data. 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.
4. Use the 1.7.2 version of DatabaseManager to create a new database, in this example 'newversion' in a different directory.
5. `SHUTDOWN` this database.
6. Copy the `newversion.script` file from step 2 over the file of the same name for the new database created in 4.
7. Try to open the new database using DatabaseManager.
8. 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.

## Manual Changes to the .script File

In 1.7.2 several ALTER TABLE commands are available to change the data structures and their names. However, if an old database cannot be opened due to data inconsistencies, or when there is no ALTER TABLE command for the particular change, manual editing of the SCRIPT file can sometimes be used.

Index and row data for CACHED tables is stored in the \*.data file. Because of this, in 1.7.2, a new command, SHUTDOWN SCRIPT, has been introduced to save all the CACHED table data in the .script file and delete the .data and \*.backup files. After issuing this command, you can make changes to the \*.script file as listed below. This procedure can also be used for adding and removing indexes or constraints to CACHED tables when the size of the \*.data file is over 1GB and the normal SQL commands do not work due to the \*.data file growing beyond 2GB.

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.
- CREATE UNIQUE INDEX . . . to CREATE INDEX . . . and vice versa

A unique index can always be converted into a normal index. A non-unique index can only be converted into a unique index if the table data for the column(s) is unique in each row.

- NOT NULL

A not-null constraint can always be removed. It can only be added if the table data for the column has no null values.

- PRIMARY KEY

A primary key constraint can be removed or added. It cannot be removed if there is a foreign key referencing the column(s).

- COLUMN TYPES

Some changes to column types are possible. For example an INTEGER column can be changed to BIGINT, or DATE, TIME and TIMESTAMP columns can be changed to VARCHAR.

Any other changes to data structures should be made only through the supported ALTER commands.

After completing the changes and saving the modified \*.script file, you can open the database as normal.

## Backing Up Databases

The data for each database consists of up to 5 files in the same directory. The endings are \*.properties, \*.script, \*.data, \*.backup and \*.log (a file with the \*.lck ending is used for controlling access to the database and should not be backed up). These should be backed up together. The files can be backed up while the engine is running but care should be taken that a CHECKPOINT or SHUTDOWN operation does not take place during the backup. It is more efficient to perform the backup immediately after a CHECKPOINT. The \*.data file can be excluded from the backup. In this case, when restoring, a dummy \*.data file is needed which can be an empty, 0 length file. The engine will expand the \*.backup file to replace this dummy file if the backup is restored. If the \*.data file is not backed up, the \*.properties file may have to be modified to ensure it contain modified=yes instead of modified=no prior to restoration. If a backup immediately follows a checkpoint, then the \*.log file can also be excluded, reducing the significant files to \*.properties, \*.script and \*.backup. Normal backup methods, such as archiving the files in a compressed bundle can be used.

---

# Chapter 5. Text Tables

## *Text Tables as a Standard Feature of Hsqldb*

Bob Preston, HSQLDB Development Group

Fred Toussi, HSQLDB Development Group <ft@cluedup.com>

Copyright 2002-2004 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 HSQLDB Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.

\$Date: 2004/08/12 20:43:20 \$

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.

HSQLDB 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 and will have wide-ranging use way beyond the currently established Java realm of HSQLDB.

### **Goals of the Implementation**

1. We aimed to finalise the DDL for Text Tables so that future releases of HSQLDB use the same DDL scripts.
2. We aimed to support Text Tables as GLOBAL TEMPORARY or GLOBAL BASE tables in the SQL domain.

## **The Implementation**

### **Definition of Tables**

Text Tables are defined similarly to conventional tables with the added TEXT keyword:

```
CREATE [TEMP] TEXT TABLE <tablename> (<column definition> [<constraint definition>])
```

In addition, a 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 memory-only databases (databases that have no script file).

## Scope and Reassignment

- A Text table without a file assigned to it is READ ONLY and EMPTY.
- A Temporary Text table has the scope and the lifetime of the SQL session (a JDBC Connection).
- 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 administrator to ensure their integrity.

From version 1.7.2 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

This has changed in 1.7.2 to support both null values and empty strings.

- 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's, VARCHARs, and LONGVARCHARs the same, the ability to assign different separators to the latter two is provided. When a different separator is assigned to a VARCHAR or LONGVARCHAR field, it will terminate any CSV field of that type. For example, if the first field is CHAR, and the second field LONGVARCHAR, 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
```

The following example shows how to change the default separator to the pipe (|), VARCHAR separator

to the period (.) and the LONGVARCHAR separator to the tilde (~). Place the following within the SET TABLE SOURCE statement, for example:

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

HSQLDB also recognises the following special indicators for separators:

### special indicators for separators

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

Furthermore, HSQLDB provides csv file support with three additional boolean options: `ignore_first`, `quoted` and `all_quoted`. (1.7.2) 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. 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 from the bottom up, by placing the keyword "DESC" at the end of the SET TABLE SOURCE statement:

```
SET TABLE mytable SOURCE "myfile" DESC
```

This feature provides functionality similar to the Unix tail command, by re-reading the file each time a select is executed. Using this feature sets the table to read-only mode. Afterwards, it will no longer be possible to change the read-only status with `SET TABLE <tablename> READONLY TRUE`.

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  $3 \cdot (2^{**scale})$ , 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
```

## Text File Issues

### 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 in the database properties file.
- Blank lines are allowed anywhere in the text file, and are ignored.
- The file location for a text table created with

```
SELECT <select list> INTO TEXT <tablename> FROM
```

is the directory that contains the database and the file name is based on the table name. The table name is converted into the file name by replacing all the non-alphanumeric characters with the underscore character, conversion into lowercase, and adding the `.csv` suffix.

- From version 1.7.2 it is possible to define a primary key or identity column 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`.
- 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 not allowed with any string that contains the newline or the carriage return character.

## Text File Global Properties

---

### **Complete list of supported global properties in \*.properties files**

- `textdb.fs`
- `textdb.lvs`
- `textdb.quoted`
- `textdb.all_quoted`
- `textdb.ignore_first`
- `textdb.encoding`
- `textdb.cache_scale`
- `textdb.allow_full_path`

## **Importing a Text Table file in to a Traditional (non-Text Table) Table**

The directory `src/org/hsqldb/sample` in your HSQLDB distribution contains a file named `load_binding_lu.sql`. This is a working SQL file which imports a pipe-delimited text file from the database's file directory into an existing normal table. You can edit a copy of this file and use it directly with SqlTool, or you can use the SQL therein as a model (using any SQL client at all).

---

# Chapter 6. TLS

## *TLS Support (a.k.a. SSL)*

Blaine Simpson, HSQLDB Development Group  
<blaine.simpson@admc.com>

\$Date: 2004/06/08 13:04:25 \$

The instructions in this document are liable to change at any time. In particular, we will be changing the method to supply the server-side certificate password.

## Requirements

### Hsqlldb 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 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 server.
- If you are running the server side, then you'll need to run a HSQLDB Server or WebServer. It doesn't matter if the underlying database instances are new, and it doesn't matter if you are making a new Server configuration or encrypting an existing Server configuration. (You can turn encryption on and off at will).
- You need a HSQLDB jar file that was built with JSSE present. If you got your HSQLDB 1.7.2 distribution from us, you are all set, because we build with Java 1.4 (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.

#### Hsqlldb TLS URL Prefixes

- jdbc:hsqlldb:hsq1s://
- jdbc:hsqlldb:https://

At this time, the latter will only work for clients running with Java 1.4.

If the server you wish to connect to is using a certificate approved by your default trust keystores, then there is nothing else to do. If not, then you need to tell Java to "trust" the server cert. (It's a slight oversimplification 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 6.1. 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/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html#CustomizingStores) [http://java.sun.com/j2se/1.4.2/docs/guide/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 6.2. Adding a certificate to the client keystore

```
keytool -import -trustcacerts -keystore trust.store -alias new_alias -file ser
```

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 6.3. Specifying your own trust store to a JDBC client

```
java -Djavax.net.ssl.trustStore=/home/blaine/trust.store -jar /path/to/hsqldb.
```

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.

N.b. 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:hsqldb://localhost` or `jdbc:hsqldb:hsqldb://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/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html>], or in the subdirectory `docs/guide/security/jsse` of your Java SE docs.

## Server-Side

Get yourself a JKS keystore containing a private key. Then set the system property `javax.net.ssl.keyStore` to the path to that file, and `javax.net.ssl.keyStorePassword` to the password of the keystore (and to the private key-- they have to be the same).

### Example 6.4. Running an Hsqldb server with TLS encryption

```
java -Djavax.net.ssl.keyStorePassword=secret -Djavax.net.ssl.keyStore=/usr/hsqldb/keystore.jks \
-cp /path/to/hsqldb.jar org.hsqldb.Server
```

(This is a single command that I have broken into 2 lines using my shell's \ line-continuation feature. In this example, I'm using a `server.properties` file so that I don't need to give arguments to specify database instances or the server endpoint).

### Caution

Specifying a password on the command-line is definitely **not secure**. It's really only appropriate when untrusted users do not have any access to your computer.

If there is any user demand, we will have a more secure way to supply the password before long.

## JSSE

If you are running Java 4.x, 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 from the software 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 6.5. Getting a pem-style private key into a JKS keystore

```
openssl pkcs8 -topk8 -outform DER -in Xpvk.pem -inform PEM -out Xpvk.pk8 -nocr
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/j2se/1.4.2/docs/guide/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, follow the instructions in the UNIX Quick Start chapter, and remember to make the init script configuration file readable only to root and to set the variables `TLS_PASSWORD` and `TLS_KEystore`.

If you are using a private server certificate, make sure to also set the trust store filepath as shown in the sample init script configuration file.

The cautionary warning above still applies. The password will be visible to any minimally competent local UNIX user who wants to see it.



---

# Chapter 7. SqlTool

## *SqlTool Manual*

Blaine Simpson, HSQLDB Development Group  
<blaine.simpson@admc.com>

\$Date: 2004/10/20 01:46:51 \$

## Purpose

This document explains how to use SqlTool, the main purpose of which is to read your SQL text file or stdin, and execute the SQL commands therein against a JDBC database. There are also a great number of features to facilitate both interactive use (such as command-line editing and aliases) and automation (such as scripting variables and SQL transaction control and error handling).

Some of the examples below use quoting which works exactly as-is for any normal UNIX shell. I have not yet tested these commands on Windows, and I doubt whether the quoting will work just like this (though it is possible). SqlTool is still a very useful tool even if you have no quoting capability at all.

This document is now updated for version 1.37 of SqlTool and 1.86 of SqlFile (the latter is the class which does most of the work for SqlTool). The startup banner will report both versions when you run SqlTool interactively. I expect this version of this document to accurately describe SqlTool for some unknown number of versions into the future.

## The Bare Minimum You Need to Know to Run SqlTool

If you want to use SqlTool, then you either have an SQL text file, or you want to interactively type in SQL commands. If neither case applies to you, then you are looking at the wrong program.

### Procedure 7.1. To run SqlTool...

1. Copy the file `sqltool.rc` from the directory `src/org/hsqldb/sample` of your HSQLDB distribution to your home directory and secure access to it if your home directory is accessible to anybody else. This file will work as-is for a Memory Only database instance; or if your target is a HSQLDB Server running on your local computer with default settings and the password for the "sa" account is blank (the sa password is blank when new HSQLDB database instances are created). Edit the file if you need to change the target Server URL, username, password, character set, JDBC driver, or TLS trust store as documented in the Authentication Setup section.
2. Find out where your `hsqldb.jar` file resides. It typically resides at `HSQLDB_HOME/lib/hsqldb.jar` where `HSQLDB_HOME` is the base directory of your HSQLDB software installation. For this reason, I'm going to use `"$HSQLDB_HOME/lib/hsqldb.jar"` as the path to `hsqldb.jar` for my examples, but understand that you need to use the actual path to your own `hsqldb.jar` file.
3. Run

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar --help
```

to see what command-line arguments are available. Note that you don't need to worry about setting the CLASSPATH when you use the `-jar` switch to `java`. Assuming that you set up your SqlTool configuration file at the default location and you want to use the HSQLDB JDBC driver, you will want to run something like

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar mem
```

for interactive use, or

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar --noinput --sql 'SQL statement;' mem
```

or

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar mem filepath1.sql...
```

where `mem` is an *urlid*, and the following arguments are paths to text SQL files. For the filepaths, you can use whatever wildcards your operating system shell supports.

The *urlid* `mem` in these commands is a key into your SqlTool configuration file, as explained in the Authentication Setup section. Since this is a Memory Only database, you can use SqlTool with this *urlid* immediately with no database setup whatsoever (however, you can't persist any changes that you make to this database). The sample `sqltool.rc` file also defines the *urlid* "localhost-sa" for a local HSQLDB Server. At the end of this section, I explain how you can load some sample data to play with, if you want to.

## Important

SqlTool does not *commit* DML changes by default. This leaves it to the user's discretion whether to commit or rollback their modifications. Remember to either run the command `commit;` before quitting SqlTool, or use the `--autoCommit` command-line switch.

Note that the `--sql` switch runs the given commands *in addition* to standard input and/or specified SQL files. If you want to run only SQL command(s) which you give on the command-line, then use the `--sql` switch *and* the `--noinput` switch.

If you put a file named `auto.sql` into your home directory, this file will be executed automatically every time that you run SqlTool interactively and without the `--noAutoFile` switch.

To use a JDBC Driver other than the HSQLDB driver, you can't use the `-jar` switch because you need to modify the classpath. You must add the `hsqldb.jar` file and your JDBC driver classes to your classpath, and you must tell SqlTool what the JDBC driver class name is. The latter can be accomplished by either using the `--driver` switch, or setting "driver" in your config file. The Authentication Setup section explains the second method. Here's an example of the first method (after you have set the classpath appropriately).

```
java org.hsqldb.util.SqlTool --driver oracle.jdbc.OracleDriver urlid
```

## Tip

If the tables of query output on your screen are all messy because of lines wrapping, the best and easiest solution is usually to resize your terminal emulator window to make it wider. (With some terms you click & drag the frame edges to resize, with others you use a menu system where you can enter the number of columns).

If you want some sample database objects and data to play with, execute the `sampledata.sql` SQL file. `sampledata.sql` resides in the `src/org/hsqldb/sample` directory of your HSQLDB distribution. Run it like this from an SqlTool session

```
\i HSQLDB_HOME/src/org/hsqldb/sample/sampledata.sql
```

where **HSQLDB\_HOME** is the base directory of your HSQLDB software installation.

For memory-only databases, you'll need to run this every time that you run SqlTool. For other (persistent) databases, the data will reside in your database until you drop the tables.

## Authentication Setup

Authentication setup is accomplished by creating a text SqlTool configuration file. In this section, when I say *configuration* or *config* file, I mean an SqlTool configuration file (aka SqlTool RC file).

The following sample `sqltool.rc` file resides in the directory `src/org/hsqldb/sample` of your HSQLDB distribution.

### Example 7.1. Sample `sqltool.rc` File

```
# $Id: sqltool.rc,v 1.11 2004/07/18 21:34:35 unsaved Exp $
# This is a sample SqlTool configuration file, a.k.a. rc file.
# You can run SqlTool right now by copying this file to your home directory
# and running
#   java -jar /path/to/hsqldb.jar mem
# This will access the first urlid definition below in order to use a
# personal Memory-Only database.
# If you have the least concerns about security, then secure access to
# your sqltool.rc file.
# See the documentation for SqlTool for various ways to use this file.
# A personal Memory-Only database.
urlid mem
url jdbc:hsqldb:mem:memdbid
username sa
password

# 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:hsqldb://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 ../hsqsqltool.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 urlid for a Postgresql database.
# You will need to put the org.postgresql.Driver class into your
# classpath.
# The postgresql jar will be named postgresql.jar (if you built Postgresql
# from source), or something like pg73b1jdbc3.jar or jdbc7.2x-1.2.jar.
# You can obtain it from a client or server Postgresql installation, or
# download it from http://jdbc.postgresql.org/download.html.
# Notice that the jar file names (other than "postgresql.jar") contain both
# the target Postgresql server version and the client-side JDBC level (which
# is determined by your client-side Java version, as explained at
# http://jdbc.postgresql.org/download.html).
# I recommend the latest production version for your JDBC version. The
# later JDBC drivers work better even with older Postgresql servers.
# (E.g. \dt won't list owners with an older driver).
# N.b.: Suse Linux 9.1 users should download a new driver from the PG site,
# since Suse distributes the 7.3 drivers with Postgresql 7.4 (why???).
# 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 ../hsqsqltool.jar...".
# Put both the HSQLDB jar and the Postgresql jar in your classpath (and
# export!) and run something like "java org.hsqldb.util.SqlTool...".
# N.b.: I notice that Postgresql is unusual in that it does not do an
# implicit commit before DDL commands. If you get an error message
# "... cannot run inside a transaction block", just run "commit;" and retry.

#urlid commerce
#url jdbc:postgresql://dbsvr2/commercedb
#username blaine
#password obscured
#driver org.postgresql.Driver
#####

#####
# Template for a TLS-encrypted HSQLDB Server.
# Remember that the hostname in hsqldb (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:hsqldb://db.admc.com:9001/lm2
#username blaine
#password asecret
#truststore /home/blaine/ca/db/db-trust.store
#####

```

You can put this file anywhere you want to, and specify the location to SqlTool by using the "--rcfile" argument. If there is no reason to not use the default location (and there are situations where you would not want to), then use the default location and you won't have to give "--rcfile" arguments to SQLTool. The default location is *sqltool.rc* in your home directory. If you have any doubt about where that is, then just run SqlTool with a phony urlid and it will tell you where it expects the configuration file to be.

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar x
```

The config file consists of stanza(s) like this:

```
urlid web
url jdbc:hsqldb:hsq://localhost
username web
password webpassword
```

These four settings are required for every urlid. (There are optional settings also, which are described a couple paragraphs down). You can have as many blank lines and comments like

```
# This comment
```

in the file as you like. The whole point is that the *urlid* that you give in your SqlTool command must match a *urlid* in your configuration file.

## Important

Use whatever facilities are at your disposal to protect your configuration file.

It should be readable, both locally and remotely, only to users who need to use the records in it to run SqlTool. On UNIX, this is easily accomplished by using `chmod/chown` commands and making sure that it is protected from anonymous remote access (like via NFS, FTP or Samba).

You can also put the following optional settings into a urlid stanza. The setting will, of course, only apply to that urlid.

charset	Sets encoding character set for input. See the Character Encoding section of the Non-Interactive section. You can, alternatively, set this for one SqlTool invocation by setting the system property <code>sqltool.charset</code> . Defaults to <code>US-ASCII</code> .
driver	Sets the JDBC driver class name. You can, alternatively, set this for one SqlTool invocation by using the SqlTool switch <code>--driver</code> . Defaults to <code>org.hsqldb.jdbcDriver</code> .
truststore	TLS trust keystore store file path as documented in the TLS chapter. You usually only need to set this if the server is using a non-publicly-certified certificate (like a self-signed self-ca'd cert).

Property and SqlTool command-line switches override settings made in the configuration file.

## Interactive

Do read the The Bare Minimum section before you read this section.

You run SqlTool interactively by specifying no SQL filepaths on the SqlTool command line. Like this.

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar urlid
```

### **Procedure 7.2. What happens when SqlTool is run interactively (using all default settings)**

1. SqlTool starts up and connects to the specified database, using your SqlTool configuration file (as explained in the Authentication Setup section).
2. SQL file `auto.sql` in your home directory is executed (if there is one),
3. SqlTool displays a banner showing the SqlTool and SqlFile version numbers and describes the different command types that you can give, as well as commands to list all of the specific commands available to you.

You exit your session by using the `"\q"` special command or ending input (like with Ctrl-D or Ctrl-Z).

### **Important**

Every command (regardless of type) and comment must begin at the beginning of a line or immediately after a comment ends with `"*/"`.

You can't nest commands or comments. You can only start new commands (and comments) after the preceding statement has been terminated. (Remember that you if you're running SqlTool interactively, you can terminate an SQL statement without executing it by entering a blank line).

(Special Commands, Buffer Commands and PL Commands always consist of just one line. Any of these commands or comments may be preceded by space characters.)

When you are typing into SqlTool, you are always typing part of the *current command*. The *buffer* is the *last SQL command*. If you're typing an SQL command, then the previous SQL command will be in the buffer, not the one you are currently typing. The current command could be any type of command, but only SQL commands get moved to the buffer when they are completed. When you type command-editing commands, the *current command* is the editing command (like `":s/tbl/table/"`), the result of which is to modify the SQL command in the buffer (which can thereafter be executed). The `":a"` command (with no argument) is special in that it takes a copy of the SQL command in the buffer and makes that the current command, leaving you in a state where you are *appending* to that *now current* command. The buffer is the zeroeth item of the SQL command history.

### **Command types**

SQL Statement	Any command that you enter which does not begin with <code>"\"</code> , <code>":"</code> , or <code>"* "</code> is an SQL Statement. The command is not terminated when you hit ENTER, like most OS shells. You terminate SQL Statements with either <code>";"</code> at the end of a line, or with a blank line. In the former case, the SQL Statement will be executed against the SQL database and the command will go into the command
---------------	--

buffer and SQL command history for editing or viewing later on. In the latter case (you end an SQL Statement with a blank line), the command will go to the buffer and SQL history, but will not be executed (but you can execute it later from the buffer).

(Blank lines are only interpreted this way when SqlTool is run interactively. In SQL files, blank lines inside of SQL statements remain part of the SQL statement).

As a result of these termination rules, whenever you are entering text that is not a Special Command, Buffer Command, or PL Command, you are always *appending* lines to an SQL Statement. (In the case of the first line, you will be appending to an empty SQL statement. I.e. you will be starting a new SQL Statement).

Special Command	Run the command "\?" to list the Special Commands. All of the Special Commands begin with "\". I'll describe some of the most useful Special Commands below.
Buffer Command	Run the command ":@" to list the Special Commands. All of the Special Commands begin with ":". Buffer commands operate upon the command "buffer", so that you can edit and/or (re-)execute previously entered commands.
PL Command	Procedural Language commands. Run the command "* ?" to list the PL Commands. All of the PL Commands begin with "* ". PL commands are for setting and using scripting variables and conditional and flow control statements like <code>* if</code> and <code>* while</code> . Using variables as command aliases (aka macros) can be a real convenience for nearly all users, so this feature will be discussed briefly in this section. More detailed explanation of PL variables and the other PL features, with examples, are covered in the Procedural Language section.

## Essential Special Commands

\?	help
\q	quit
\dt [filter_substring] \dv [filter_substring] \d* [filter_substring] \ds [filter_substring] \da [filter_substring]	Lists available table-like objects of the given type. <ul style="list-style-type: none"> <li>• t: non-system Table#</li> <li>• v: Views</li> <li>• *: all table-like objects</li> <li>• s: System table#</li> <li>• a: database SQL Aliases</li> </ul> <p>If your database supports schemas, then the schema name will also be listed.</p>

If you supply an optional *filter substring*, then only items which contain the given substring (in the object name or schema name) will be listed. The substring test is case-insensitive.

I do intent to change "\ds" to "\dS". I would do it right now if the SqlTool code weren't going production in two days. The reason is, I want \ds for listing either Sequences (like psql) or Synonyms.

\d objectname Lists names of all columns in the specified table or view.

## Tip

When working with real data (as opposed to learning or playing), I often find it useful to run two SqlTool sessions in two side-by-side terminal emulator windows. I do all of my real work in one window, and use the other mostly for \d commands. This way I can refer to the data dictionary while writing SQL commands, without having to scroll.

\s Shows the SQL command history. The SQL command history will show a number (a negative number) for each SQL Statement that has made it into the buffer so fare (by either executing or entering a blank line). You can then use the "\-" command (which is described next) to retrieve commands from the SQL history to work with. To list just the very last command, you would use the ":!" buffer command to list the buffer contents, instead of this command.

\-[3] Enter "\" followed by the command number from SQL history, like "\-3". That command will be written to the buffer so that you can execute it or edit it using buffer commands.

(You can append a semicolon to a recall command in order to execute the recalled buffer immediately, like "\-3;". This is actually just a shortcut for running the Special Command "\-3" and the Buffer Command ";;".)

This list here includes only the *essential* Special Commands, but n.b. that there are other useful Special Commands which you can list by running \?. (You can, for example, execute SQL from external SQL files, and save your interactive SQL commands to files). Some specifics of these other commands are specified immediately below, and the Generating Text or HTML Reports section explains how to use the "\o" and "\H" special commands to generate reports.

Be aware that the \! Special Command does not work for external programs that read from standard input. You can invoke non-interactive and graphical interactive programs, but not command-line interactive programs.

SqlTool executes \! programs directly, it does not run an operating system shell (this is to avoid OS-specific code in SqlTool). Because of this, you can give as many command-line arguments as you wish, but you can't use shell wildcards or redirection.

The \w command can be used to store any command in your SQL history to a file. Just restore the command to the buffer (which is the 0th element of the history) with a command like "\-4" before you give the \w command.

## Buffer Commands



:? help

:: Executes the SQL statement in the current buffer against the database.

:l (This is a lower case L). List the current contents of the buffer.

:a Enter append mode with the contents of the buffer as the current SQL Statement. Things will be exactly as if you physically re-typed the command that is in the buffer. Whatever line you type next will be appended to the SQL Statement. You can execute the command by terminating a line with ";", or send it back to the buffer by entering a blank line.

You can, optionally, put a string after the :a, in which case this text will be appended and you will remain in append mode. (Unless the text ends with ';', in which case the resultant statement will be executed immediately). Note that if you do put text after the "a", *exactly* what you type immediately after "a" will be appended. If your buffer contains `SELECT x FROM mytab` and you run `a:le`, the resultant command will be `SELECT x FROM mytable`. If your buffer contains `SELECT x FROM mytab` and you run `a: ORDER BY y`, the resultant command will be `SELECT x FROM mytab ORDER BY y`. Notice that in the latter case the append text begins with a space character.

:s/from string/to string/switches This is the primary command for SqlTool command editing-- it operates upon the current buffer. The "to string" and the "switches" are both optional. To start with, I'll discuss the use and behavior if you don't supply any substitution mode switches.

Don't use "/" if it occurs in either "from string" or "to string". You can use any character that you want in place of "/", but it must not occur in the *from* or *to* strings. Example

```
:s@from string@to string@
```

The *to* string is substituted for the first occurrence of the (case-specific)*from* string. The replacement will consider the entire SQL statement, even if it is a multi-line statement.

All occurrences of "\$" in the *from* string and the *to* string are treated as line breaks. For example, *from* string of "\$FROM mytable" would actually look for occurrences of

```
*
FROM mytable
```

Here is a another meaningful example using \$.

```
:s/e)/e) WHERE coll is not null$/
```

This command appends "WHERE coll is not null" to the line(s) which end with "e)".

The *to* string may be empty, in which case, occurrences of the *from* string are just deleted. For example

```
:s/this//
```

would remove the first occurrence of "this". (With the "g" substitution mode switch, as explained below, it would remove all occurrences of "this").

Don't end a *to* string with ";" in attempt to make a SQL statement execute. There is a substitution mode switch to use for that purpose.

You can use any combination of the substitution mode switches.

- Use "i" to make the searches for *from* string case insensitive.
- Use "g" to substitute globally, i.e., for all occurrences of *from* string which are found in the text under consideration.
- Use ";" to execute the command immediately after the substitution is performed.
- Use an integer (from 1 to 9) to narrow the text under consideration to a specific line of a multi-line buffer.

The substitution facility doesn't support any regular expressions at all. When we stop supporting Java versions older than 1.4, I'll start supporting regular expressions and other advanced string manipulation functions.

## Essential PL Command

\* VARNAME = value      Set the value of a variable. If the variable doesn't exist yet, it will be created.

If you set a variable to an SQL statement (without the terminating ";") you can then use it as an alias like \*VARNAME, as shown in this example.

### Example 7.2. Defining and using an alias (PL variable)

```
* q = SELECT COUNT(*) FROM mytable  
*q;  
*q WHERE mass > 200;
```

If you put variable definitions into the SQL file `auto.sql` in your home directory, those aliases/variables will always be available for interactive use.

See the Procedural Language section below for information on using variables in other ways, and information on the other PL commands and features.

## SQL History

The SQL history shown by the `\s` command, and used by other commands, is truncated to 20 entries, since the utility comes from being able to quickly view the history list. You can change the history length by setting the system property `sqltool.historyLength` to an integer like

```
java -Dsqltool.historyLength=40 -jar $HSQLDB_HOME/lib/hsqldb.jar urlid
```

The SQL history list explicitly does not contain Special, Buffer, or PL commands. It only contains SQL commands, valid or invalid, successful or unsuccessful. The reason for including bad SQL commands is so that you can recall and edit them if you want to. The same applies to the editing buffer (which is element 0 of the history).

## Raw mode

If for some reason you want SqlTool to process your commands as if it were reading an SQL file (i.e., no startup banner, no command prompts, no Buffer/history commands, aborting upon failure by default, allow blank lines with SQL statements), then specify a SQL filepath of "-", like

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar urlid -
```

This is very useful in (at least) two situations. This is how you would pipe input into SqlTool, which is extremely useful in shell scripts where the SQL code is generated dynamically, like

### Example 7.3. Piping input into SqlTool

```
echo "Some SQL commands with '$VARIABLES';" |  
java -jar $HSQLDB_HOME/lib/hsqldb.jar urlid -
```

This is also a good way to test what SqlTool will do when it encounters a specific command in a SQL file. You *emulate* SQL file execution while giving SqlTool commands interactively.

## Non-Interactive

Read the Interactive section if you have not already, because much of what is in this section builds upon that. Even if your plans are to run SqlTool non-interactively, you should really learn to run it interactively because it's such a powerful debugging tool, and you can use it to prototype sql scripts.

### Important

If you're doing data updates, remember to issue a commit command or use the `-autoCommit` switch.

As you'll see, SqlTool has many features that are very convenient for scripting. But what really makes it superior for automation tasks (as compared to SQL tools from other vendors) is the ability to reliably de-

tect errors and to control JDBC transactions.

## Giving SQL on the Command Line

If you just have a couple SQL commands to run, you can run them directly from the command-line or from a shell script without an SQL file, like this.

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar --noinput --sql 'SQL statement;' urlid
```

Since SqlTool executes SQL statements only when a statement line is terminated with ";", you can only execute more than one SQL statement this way if your OS shell has some mechanism to pass linebreaks in arguments through to the target program. With any Bourne-compatible shell, you can include linebreaks in the SQL statements like this.

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar --noinput --sql 'SQL statement;' urlid '
SQL statement number one;
SQL statement
number two;
SQL statement three;
' urlid
```

If you don't give the `--noinput` switch, then after executing the given statements, an interactive session will be started.

The `--sql` switch is very useful for setting shell variables to the output of SQL Statements, like this.

```
# A shell script
USERCOUNT=`java -jar $HSQLDB_HOME/lib/hsqldb.jar --noinput --sql 'select count
# Handle the SqlTool error
}'
echo "There are $USERCOUNT users registered in the database."
[ "$USERCOUNT" -gt 3 ] && { # If there are more than 3 users registered
# Some conditional shell scripting
```

## SQL Files

Just give paths to sql text file(s) on the command line after the *urlid*.

Often, you will want to redirect output to a file, like

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar sql... > /tmp/log.sql 2>&1
```

(Skip the "2>&1" if you're on Windows).

You can also execute SQL files from an interactive session with the "`\i`" Special Command, but be aware that the default behavior in an interactive session is to continue upon errors. If the SQL file was written without any concern for error handling, then the file will continue to execute after errors occur. You could run `\* false` before `\i filename`, but then your SqlTool session will exit if an error is encountered in the SQL file. If you have an SQL file without error handling, and you want to abort that file when an error occurs, but not exit SqlTool, the easiest way to accomplish this is usually to add `\* false` to the top of the script.

If you specify multiple SQL files on the command-line, the default behavior is to exit SqlTool if any of the SQL files encounters an error.

**SQL files themselves have ultimate control over error handling.** Regardless of what command-line options are set, or what commands you give interactively, if a SQL file gives error handling statements, they will take precedence.

You can also use `!i` in SQL files. This results in nested SQL files.

You can use the following SQL file, `sample.sql`, which resides in the `src/org/hsqldb/sample` directory of your HSQLDB distribution. It contains SQL as well as Special Commands making good use of most of the Special Commands documented below.

```
/*
    $Id: sample.sql,v 1.3 2004/07/12 16:51:01 unsaved Exp $
    Exemplifies use of SqlTool.
    PCTASK Table creation
*/

/* Ignore error for the two drop statements
 * For HSQLDB databases, you can use "IF EXISTS" instead of ignoring errors:
 *   DROP TABLE x IF EXISTS;
 * "IF EXISTS" is non-portable, however.
 */
\* true
DROP TABLE pctasklist;
DROP TABLE pctask;
\* false

\p Creating table pctask
CREATE TABLE pctask (
    id integer identity,
    name varchar(40),
    description varchar,
    url varchar,
    UNIQUE (name)
);

\p Creating table pctasklist
CREATE TABLE pctasklist (
    id integer identity,
    host varchar(20) not null,
    tasksequence int not null,
    pctask integer,
    assigndate timestamp default current_timestamp,
    completedate timestamp,
    show bit default true,
    FOREIGN KEY (pctask) REFERENCES pctask,
    UNIQUE (host, tasksequence)
);

\p Granting privileges
GRANT select ON pctask TO public;
GRANT all ON pctask TO tomcat;
GRANT select ON pctasklist TO public;
GRANT all ON pctasklist TO tomcat;

\p Inserting test records
INSERT INTO pctask (name, description, url) VALUES (
    'task one', 'Description for task 1', 'http://cnn.com');
INSERT INTO pctasklist (host, tasksequence, pctask) VALUES (
    'admc-masq', 101, SELECT id FROM pctask WHERE name = 'task one');
```

```
commit;
```

You can execute this SQL file with a Memory Only database with a command like

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar --sql "create user 'tomcat' password 'x
```

(The `--sql "create..."` arguments create an account which the script uses).

## Optimally Compatible SQL Files

If you want your SQL scripts optimally compatible among other SQL tools, then don't use any Special or PL Commands. SqlTool has default behavior which I think is far superior to the other SQL tools, but you will have to disable these defaults in order to have optimally compatible behavior.

These switches provide compatibility at the cost of poor control and error detection.

- `--continueOnErr`

The output will still contain error messages about everything that SqlTool doesn't like (malformatted commands, SQL command failures, empty SQL commands), but SqlTool will continue to run. Errors will not cause rollbacks (but that won't matter because of the following setting).

- `--autoCommit`

You don't have to worry about accidental expansion of PL variables, since SqlTool will never expand PL variables if you don't set any variables on the command line, or give any `"*"` PL commands. (And you could not have `"*"` commands in a compatible SQL file).

## Comments

SQL comments of the form `/*...*/` must begin where a (SQL/Special/Buffer/PL) Command could begin, and they end with the very first `"*/"` (regardless of quotes, nesting, etc. You may have as many blank lines as you want inside of a comment).

### Example 7.4. Valid comment example

```
SELECT count(*) FROM atable;
/* Lots of
   comments interspersed among
   several lines */  SELECT count(*)
FROM btable;
```

Notice that a command can start immediate after the comment ends.

### Example 7.5. Invalid comment example

```
SELECT count(*) FROM
/* atable */
btable;
```

This comment is invalid because you could not start another command at the comment location (because it is within an SQL Statement).

You can try using `/* . . . */` in other locations, and `--` style SQL comments, but SqlTool will not treat them as comments. If they occur within an SQL Statement, SqlTool will pass them to the database engine, and the DB engine will determine whether to parse them as comments.

## Special Commands and Buffer Commands in SQL Files

Don't use Buffer Commands in your sql files, because they won't work. Buffer Commands are for interactive use only.

`\q` [abort message]

Be aware that the `\q` command will cause SqlTool to completely exit. If a script `x.sql` has a `\q` command in it, then it doesn't matter if the script is executed like

```
java -jar ../hsqldb.jar urlid a.sql x.sql z.sql
```

or if you use `\i` to read it in interactively, or if another SQL file uses `\i` to nest it. If `\q` is encountered, SqlTool will quit. See the Procedural Language section for commands to abort an SQL file (or even parts of an SQL file) without causing SqlTool to exit.

`\q` takes an optional argument, which is an abort message. If you give an abort message, the message is displayed to the user and SqlTool will exit with a failure status. If you give no abort message, then SqlTool will exit quietly with successful status.

`\p` [text to print]

Print the given string to stdout. Just give `"\p"` alone to print a blank line.

`\i` /path/to/file.sql

Include another SQL file at this location. You can use this to nest SQL files. For database installation scripts I often have a master SQL file which includes all of the other SQL files in the correct sequence. Be aware that the current continue-upon-error behavior will apply to included files until such point as the SQL file runs its own error handling commands.

`\H`

Toggle HTML output mode. If you redirect output to a file, this can make a long session log much easier to view. This will HTML-ify the entire session. For example,

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar urlid filepath1.sql...
```

(See the Generating Text or HTML Reports section about how to easily store just the query output to file.)

`\a` [true|false]

This turns on and off SQL transaction autocommits. Auto-commit defaults to false, but you can change that behavior by using the `-autoCommit` command-line switch.

`\* [true|false]`

A "true" setting tells SqlTool to continue when errors are encountered. The current transaction will not be rolled back upon SQL errors, so if `\*` is true, then run the `ROLLBACK;` command yourself if that's what you want to happen. The default for interactive use is to continue upon error, but the default for non-interactive use is to abort upon error. You can override this behavior by using the `--continueOnErr` or the `-abortOnErr` command-line switch.

With database setup scripts, I usually find it convenient to set "true" before dropping tables (so that things will continue if the tables aren't there), then set it back to false so that real errors are caught. `DROP TABLE tablename IF EXISTS;` is a more elegant, but less portable, way to accomplish the same thing.

## Tip

It depends on what you want your SQL files to do, of course, but I usually want my SQL files to abort when an error is encountered, without necessarily killing the SqlTool session. If this is the behavior that you want, then put an explicit `\* false` at the top of your SQL file and turn on continue-upon-error only for sections where you really want to permit errors, or where you are using PL commands to handle errors manually. This will give the desired behavior whether your script is called by somebody interactively, from the SqlTool command-line, or included in another SQL file (i.e. nested).

## Important

The default settings are usually best for people who don't want to put in any explicit `\*` or error handling code at all. If you run SQL files from the SqlTool command line, then any errors will cause SqlTool to roll back and abort immediately. If you run SqlTool interactively and invoke SQL files with `\i` commands, the scripts will continue to run upon errors (and will not roll back). This behavior was chosen because there are lots of SQL files out there that produce errors which can be ignored; but we don't want to ignore errors that a user won't see. I reiterate that any and all of this behavior can (and often should) be changed by Special Commands run in your interactive shell or in the SQL files. Only you know whether errors in your SQL files can safely be ignored.

## Automation

SqlTool is ideal for mission-critical automation because, unlike other SQL tools, SqlTool returns a dependable exit status and gives you control over error handling and SQL transactions. Autocommit is off by default, so you can build a completely dependable solution by intelligently using `\*` commands (Continue upon Errors) and commit statements, and by verifying exit statuses.

Using the SqlTool Procedural Language, you have ultimate control over program flow, and you can use variables for database input and output as well as for many other purposes. See the Procedural Language section.

## Getting Interactive Functionality with SQL Files

Some script developers may run into cases where they want to run with sql files but they also want SqlTool's interactive behavior. For example, they may want to do command recall in the sql file, or they may want to log SqlTool's command-line prompts (which are not printed in non-interactive mode). In this case, do not give the sql file(s) as an argument to SqlTool, but pipe them in instead, like



```
java -jar $HSQLDB_HOME/lib/hsqldb.jar urlid < filepath1.sql > /tmp/log.html 2>&1  
or
```

```
cat filepath1.sql... |  
java -jar $HSQLDB_HOME/lib/hsqldb.jar urlid > /tmp/log.html 2>&1
```

## Character Encoding

SqlTool defaults to the US-ASCII character set (for reading). You can use another character set by setting the system property `sqltool.charset`, like

```
java -Dsqltool.charset=UTF-8 -jar $HSQLDB_HOME/lib/hsqldb.jar urlid filepath1.sql.
```

You can also set this per urlid in the SqlTool configuration file. See the Authentication Setup section about that.

## Generating Text or HTML Reports

This section is about making a file containing the output of database queries. You can generate reports by using operating system facilities such as redirection, tee, and cutting and pasting. But it is much easier to use the `\o` and `\H` special commands.

### Procedure 7.3. Writing query output to an external file

1. By default, everything will be done in plain text. If you want your report to be in HTML format, then give the special command `\H`. If you do so, you will probably want to use filenames with an suffix of `.html` or `.htm` instead of `.txt` in the next step.
2. Run the command `\o path/to/reportfile.txt`. From this point on, output from your queries will be appended to the specified file. (I.e. another *copy* of the output is generated.) This way you can continue to monitor or use output as usual as the report is generated.
3. When you want SqlTool to stop writing to the file, run `\o` (or just quit SqlTool if you have no other work to do).
4. If you turned on HTML mode with `\H` before, you can run `\H` again to turn it back off, if you wish.

It is not just the output of "SELECT" statements that will make it into the report file, but

### Kinds of output that get teed to `\o` files

- Output of SELECT statements.
- Output of all `\d` Special Commands. (I.e., `\dt`, `\dv`, etc., and `\d OBJECTNAME`).
- Output of `\p` Special Commands. You will want to use this to add titles, and perhaps spacing, for

the output of individual queries.

Other output will go to your screen or stdout, but will not make it into the report file. Be aware that no error messages will go into the report file. If SqlTool is run non-interactively (including if you give any SQL file(s) on the command line), SqlTool will abort with an error status if errors are encountered. The right way to handle errors is to check the SqlTool exit status. (The described error-handling behavior can be modified with SqlTool command-line switches and Special Commands).

## Warning

Remember that `\o` *appends* to the named file. If you want a new file, then use a new file name or remove the target file ahead of time.

## Tip

So that I don't end up with a bunch of junk in my report file, I usually leave `\o` off while I perfect my SQL. With `\o` off, I perfect the SQL query until it produces on my screen exactly what I want saved to file. At this point I turn on `\o` and run `":"` to repeat the last SQL command. If I have several complex queries to run, I turn `\o` off and repeat until I'm finished. (Every time you turn `\o` on, it will append to the file, just like we need).

Usually it doesn't come to mind that I need a wider screen until a query produces lines that are too long. In this case, stretch your window and repeat the last command with the `":"` Buffer Command.

# Procedural Language

Aka PL

Most importantly, run `SqlTool` interactively and give the `"* ?"` command to see what PL commands are available to you.

PL variables will only be expanded after you run a PL command (or set variable(s) from the command-line). We only want to turn on variable expansion if the user wants variable expansion. People who don't use PL don't have to worry about strings getting accidentally expanded.

For all PL commands (other than plain `"*"`) the leading `"*"` must be followed by whitespace. The only reason for this requirement is to provide for aliases, i.e., the ability to short-cut `*VARIABLE` for `*{VARIABLE}` at the beginning of a command. Therefore, `"*X"` is an alias whereas `"* X"` is a PL command.

The command `"*"` is an exception to this rule. it doesn't matter whether you put white space after it. All other PL commands imply the `"*"` command, so you only need to use the `"*"` statement if your script uses PL variables and it is possible that no variables may be set before-hand (and no PL commands have been run previously). In this case, without `"*"`, your script would silently use a literal value like `"*{x}"` instead of trying to expand it. With a preceding `"*"` command, PL will notice that the variable `x` has not been set and will generate an error. (If `x` had been set here will be no issue because setting a variable automatically turns on PL variable expansion).

Aliasing may only be used for SQL statements. You can define variables for everything in a Special or PL Command, except for the very first character ("`\`" or `"*"`). Therefore, you can use variables but not aliasing in Special and PL Commands. Here is a hyperbolically impractical example to show the extent to which PL variables can be used in Special commands even though you can not use them as aliases.

```
sql> * qq = p Hello Butch
sql> \*{qq} done now
Hello Butch done now
```

## Variables

- Use the `* list` command to list some or all variables and their values.
- You can set variables using the `* VARNAME = value` command.
- You can also set variables using the `--setvar` command-line switch. I give a very brief but useful example of this below.
- Variables are always expanded in SQL, Special, and PL commands if they are written like `*{VARNAME}` (assuming that a PL command has been run previously). Your SQL scripts can give good feedback by echoing the value of variables with the `"\p"` special command.
- Variables written like `*VARNAME` are expanded if they *begin* an SQL Statement. (They must also be followed by whitespace or terminate the Statement). I usually refer to this use of PL variables as *aliasing*. If the value of a variable is an entire SQL command, you generally do not want to include the terminating `;"` in the value. There is an example of this above.
- Variables are normally written like `*VARNAME` in logical expressions to prevent them from being evaluated too early. See below about logical expressions.
- You can't do math with expression variables, but you can get functionality like the traditional `for (i = 0; i < x; i++)` by appending to a variable and testing the string length, like

```
* while (*i < ${x})
  * i = *{i}.
```

`i` will be a growing line of dots.

Here is a short SQL file that gives the specified user write permissions on some application tables.

### Example 7.6. Simple SQL file using PL

```
/*
 grantwrite.sql

 Run SqlTool like this:
   java -jar path/to/hsqldb.jar -setvar USER=debbie grantwrite.sql
*/

/* Explicitly turn on PL variable expansion, in case no variables have
   been set yet. (Only the case if user did not set USER).
*/
*

GRANT all ON book TO *{USER};
GRANT all ON category TO *{USER};
```

Note that this script will work for any (existing) user just by supplying a different user name on the command-line. I.e., no need to modify the tested and proven script. There is no need for a `commit` statement in this SQL file since no DML is done. If the script is accidentally run without setting the `USER` variable, SqlTool will give a very clear notification of that.

The purpose of unsetting the INIT variable is just to initialize PL so that the `*{USER}` variables will be expanded. (This would not be necessary if the USER variable, or any other variable, were set, but we don't want to depend upon that).

## Logical Expressions

Logical expressions occur only inside of logical expression parentheses in PL statements. For example, `if (*var1 > astring)` and `while (*checkvar)`. (The parentheses after "foreach" do not enclose a logical expression, they just enclose a list).

There is a critical difference between `*{VARNAME}` and `*VARNAME` inside logical expressions. `*{VARNAME}` is expanded one time when the parser first encounters the logical expression. `*VARNAME` is re-expanded every time that the expression is evaluated. So, you would never want to code `* while (*{X} < 5)` because the statement will always be true or always be false. (I.e. the following block will loop infinitely or will never run).

Don't use quotes or whitespace of any kind in `*{VARNAME}` variables in expressions. (They would expand and then the expression would most likely no longer be a valid expression as listed in the table below). Quotes and whitespace are fine in `*VARNAME` variables, but it is the entire value that will be used in evaluations, regardless of whether quotes match up, etc. I.e. quotes and whitespace are not *special* to the token evaluator.

### Logical Operators

TOKEN	The token may be a literal, a <code>*{VARNAME}</code> which is expanded early, or a <code>*VARNAME</code> which is expanded late. (You usually do not want to use <code>*{VARNAME}</code> in logical expressions). False if the token is not set, empty, or "0". True otherwise.
TOKEN1 == TOKEN2	True if the two tokens are equivalent "strings".
TOKEN1 <> TOKEN2	Ditto.
TOKEN1 >< TOKEN2	Ditto.
TOKEN1 > TOKEN2	True if the TOKEN1 string is longer than TOKEN2 or is the same length but is greater according to a string sort.
TOKEN1 < TOKEN2	Similarly to TOKEN1 > TOKEN2.
! LOGICAL_EXPRESSION	Logical negation of any of the expressions listed above.

`*VARNAME`s in logical expressions, where the VARNAME variable is not set, evaluate to an empty string. Therefore `(*UNSETVAR = 0)` would be false, even though `(*UNSETVAR)` by itself is false and `(0)` by itself is false.

When developing scripts, you definitely use SqlTool interactively to verify that SqlTool evaluates logical expressions as you expect. Just run `* if` commands that print something (i.e. `\p`) if the test expression is true.

## Flow Control

Flow control works by conditionally executing blocks of Commands according to conditions specified by logical expressions.

The conditionally executed blocks are called *PL Blocks*. These PL Blocks always occur between a PL flow control statement (like `* foreach`, `* while`, `* if`) and a corresponding `* end PL Command` (like `* end foreach`).

## Caution

Be aware that the PL block reader is ignorant about SQL statements and comments when looking for the end of the block. It just looks for lines beginning with some specific PL commands. Therefore, if you put a comment line before a PL statement, or if a line of a multi-line SQL statement has a line beginning with a PL command, things may break.

I am not saying that you shouldn't use PL commands or SQL commands inside of PL blocks--you definitely should! I'm saying that in PL blocks you should not have lines inside of SQL statements or comments which could be mistaken for PL commands. (Especially, "commenting out" PL end statements will not work if you leave `* end` at the beginning of the line).

(This limitation will very likely be removed in a future version of SqlTool).

The values of control variables for `foreach` and `while` PL blocks will change as expected.

There are `* break` and `* continue`, which work as any shell scripter would expect them to. The `* break` command can also be used to quit the current SQL file without triggering any error processing. (I.e. processing will continue with the next line in the *including* SQL file or interactive session, or with the next SQL file if you supplied `multiple` on the command-line).

Below is an example SQL File that shows how to use most PL features. If you have a question about how to use a particular PL feature, check this example before asking for help. This file resides in the `src/org/hsqldb/sample` directory with the name `pl.sql`. Definitely give it a run, like

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar mem $HSQLDB_HOME/src/org/hsqldb/sample/pl.java
```

### Example 7.7. SQL File showing use of most PL features

```
/*
  $Id: pl.sql,v 1.3 2004/06/10 01:44:52 unsaved Exp $
  SQL File to illustrate the use of SqlTool PL features.
  Invoke like
      java -jar ../hsqldb.jar ../pl.sql mem
                                          -- blaine
*/

* if (! *MYTABLE)
  \p MYTABLE variable not set!
  /* You could use \q to Quit SqlTool, but it's often better to just
     break out of the current SQL file.
     If people invoke your script from SqlTool interactively (with
     \i yourscripename.sql) any \q will kill their SqlTool session. */
  \p Use arguments "--setvar MYTABLE=mytablename" for SqlTool
  * break
* end if

/* Turning on Continue-upon-errors so that we can check for errors ourselves.*/
\* true

\p
\p Loading up a table named '*{MYTABLE}'...
```

```
/* This sets the PL variable 'retval' to the return status of the following
   SQL command */
* retval ~
CREATE TABLE *{MYTABLE} (
    i int,
    s varchar
);
\p CREATE status is *{retval}
\p

/* Validate our return status. In logical expressions, unset variables like
   *unsetvar are equivalent to empty string, which is not equal to 0
   (though both do evaluate to false on their own, i.e. (*retval) is false
   and (0) is false */
* if (*retval != 0)
    \p Our CREATE TABLE command failed.
    * break
* end if

/* Default Continue-on-error behavior is what you usually want */
/* false
\p

/* Insert data with a foreach loop.
   These values could be from a read of another table or from variables
   set on the command line like
*/
\p Inserting some data into our new table (you should see 3 row update messages)
* foreach VALUE (12 22 24 15)
    * if (*VALUE > 23)
        \p Skipping *{VALUE} because it is greater than 23
        * continue
        \p YOU WILL NEVER SEE THIS LINE, because we just 'continued'.
    * end if
    INSERT INTO *{MYTABLE} VALUES (*{VALUE}, 'String of *{VALUE}');
* end foreach
\p

* themax ~
/* Can put Special Commands and comments between "* VARNAME ~" and the target
   SQL statement. */
\p We're saving the max value for later. You'll still see query output here:
SELECT MAX(i) FROM *{MYTABLE};

/* This is usually unnecessary because if the SELECT failed, retval would
   be undefined and the following print statement would make SqlTool exit with
   a failure status */
* if (! *themax)
    \p Failed to get the max value.
    /* It's possible that the query succeeded but themax is "0".
       You can check for that if you need to. */
    * break
    \p YOU WILL NEVER SEE THIS LINE, because we just 'broke'.
* end if

\p
\p #####
\p The results of our work:
SELECT * FROM *{MYTABLE};
\p MAX value is *{themax}

\p
\p Everything worked.
```

## Using `hsqsqltool.jar`

This section is only for those users who want to use SqlTool but without the overhead of `hsqldb.jar`.

If you do not need to directly use JDBC URLs like `jdbc:hsqldb:mem: + something`, `jdbc:hsqldb:file: + something`, or `jdbc:hsqldb:res: + something`, then you can use `hsqsqltool.jar` in place of the much larger `hsqldb.jar` file. `hsqsqltool.jar` will work for all JDBC databases other than HSQLDB Memory-only and In-process databases (the latter are fine if you access them via a HSQLB Server or WebServer). You will have to supply the JDBC driver for non-HSQLDB URLs.

The HSQLDB distribution doesn't "come with" a pre-built `hsqsqltool.jar` file. I recommend that you either download the Java 1.4 jar at <http://admc.com/dist/hsqsqltool-1.7.2.jar> (by right-clicking and downloading if you're reading this with a web browser), or build the `jarsqltool` target, as explained in the Building Hsqldb version 1.7.2 appendix.

If you are using the HSQLDB JDBC driver (i.e., you're connecting up to a URL like `jdbc:hsqldb:hsql + something` or `jdbc:hsqldb:http + something`), you run SqlTool exactly as with `hsqldb.jar` except you use the file path to your `hsqsqltool.jar` file (for example, `hsqsqltool-1.7.2.jar`) instead of the path to `hsqldb.jar`.

If you are using a non-HSQLDB JDBC driver, follow the instructions at the end of the The Bare Minimum section, but use your `hsqsqltool.jar` file in place of `hsqldb.jar`.

## Unit Testing SqlTool

A unit testing framework is in place. This assures the robustness of SqlTool. See the file `testrun/sqltool/readme.txt` for instructions on running, modifying, or creating unit tests. To create a new unit test, you create a SQL file and embed metacommands in the SQL file inside of comments. The metacommands tell the test harness (`org.hsqldb.test.SqlToolHarness`) how to run SqlTool (like with what arguments) and what output to expect (i.e. the test criteria). You can run tests without JUnit, or you can make a JUnit wrapper in the normal fashion. Any SQL test file can be added to our JUnit SqlTool test suite by just adding the SQL file name and description to the `testrun/sqltool/*.list` file for the desired JUnit test method.

(The SqlTool unit tests require java 1.4).

---

# Chapter 8. SQL Syntax

The Hypersonic SQL Group

Fred Toussi, HSQLDB Development Group <ft@cluedup.com>

Peter Hudson, HSQLDB Development Group

Joe Maher, HSQLDB Development Group

<jrmaher@ameritech.net>

Edited by Blaine Simpson

\$Date: 2004/11/24 23:06:37 \$

HSQLDB version 1.7.2 supports the SQL statements and syntax described in this chapter.

## Notational Conventions Used in this Chapter

[A] means A is optional.

{ B | C } means either B or C must be used.

[ { B | C } ] means either B or C may optionally be used, or nothing at all.

( and ) are the actual characters '(' and ')' used in statements.

UPPERCASE words are keywords

## SQL Commands

### ALTER INDEX<sup>1</sup>

```
ALTER INDEX <indexname> RENAME TO <newname>;
```

Index names can be changed so long as they do not conflict with other user-defined or system-defined names.

### ALTER SEQUENCE<sup>1</sup>

```
ALTER SEQUENCE <sequencename> RESTART WITH <value>;
```

Resets the next value to be returned from the sequence.

### ALTER TABLE<sup>1</sup>

```
ALTER TABLE <tablename> ADD [COLUMN] <columnname> Datatype [(columnSize[,precision  
[DEFAULT <defaultValue> [NOT NULL]] [BEFORE <existingcolumn>]);
```

<sup>1</sup> These features were added by HSQL Development Group since April 2001



Adds the column to the end of the column list. Optional attributes, size and default value (with or without NOT NULL) can be specified. The optional BEFORE <existingcolumn> can be used to specify the name of an existing column so that the new column is inserted in a position just before the <existingcolumn>. If NOT NULL is specified and the table is not empty, then a default value must be specified.

If an SQL view includes a SELECT \* FROM <tablename> in its select statement, the new column is added to the view. This is a non-standard feature which is likely to change in the future.

```
ALTER TABLE <tablename> DROP [COLUMN] <columnname>;
```

Drops the column from the table. Will not work if column is part of a primary key, unique or foreign key constraint.

This will fail if an SQL view includes the column.

```
ALTER TABLE <tablename> ALTER COLUMN <columnname> {RENAME TO <newname> | DROP
```

Changes column name. Or adds or drops the default value.

```
ALTER TABLE <tablename> ADD [CONSTRAINT <constraintname>] CHECK (<search condi
```

Adds a check constraint to the table. In the current version, a check constraint can reference only the row being inserted or updated.

```
ALTER TABLE <tablename> ADD [CONSTRAINT <constraintname>] UNIQUE (<column list
```

Adds a unique constraint to the table. This will not work if there is already a unique constraint covering exactly the same <column list>.

This will work only if the values of the column list for the existing rows are unique or include a null value.

```
ALTER TABLE <tablename> ADD [CONSTRAINT <constraintname>] FOREIGN KEY (<column  
[ON {DELETE | UPDATE} {CASCADE | SET DEFAULT | SET NULL}];
```

Adds a foreign key constraint to the table, using the same constraint syntax as when the foreign key is specified in a table definition.

This will fail if for each existing row in the referring table, a matching row (with equal values for the column list) is not found in the referenced tables.

```
ALTER TABLE <tablename> DROP CONSTRAINT <constraintname>;
```

Drops a named unique, check or foreign key constraint from the table.

```
ALTER TABLE <tablename> RENAME TO <newname>;
```

## ALTER USER<sup>1</sup>

```
ALTER USER <username> SET PASSWORD <password>;
```

Changes the password for an existing user.

Only an administrator can do this.

## CALL

```
CALL Expression;
```

Any expression can be called like a stored procedure, including, but not only Java stored procedures or functions. This command returns a ResultSet with one column and one row (the result) just like a SELECT statement with one row and one column.

See also: Stored Procedures / Functions, SQL Expression.

## CHECKPOINT

```
CHECKPOINT [DEFRAG1];
```

Closes the database files, rewrites the script file, deletes the log file and opens the database.

If DEFRAG is specified, this command also shrinks the .data file to its minimal size.

See also: SHUTDOWN, SET LOGSIZE.

## COMMIT

```
COMMIT [WORK];
```

Ends a transaction and makes the changes permanent.

See also: ROLLBACK, SET AUTOCOMMIT, SET LOGSIZE.

## CONNECT

```
CONNECT USER <username> PASSWORD <password>;
```

Connects to the database as a different user. Use "" for an empty password.

See also: GRANT, REVOKE.

## CREATE ALIAS

```
CREATE ALIAS <function> FOR <javaFunction>;
```

Creates an alias for a Java function. The function must be accessible from the JVM in which the database runs. Example:

```
CREATE ALIAS ABS FOR "java.lang.Math.abs";
```

See also: CALL, Stored Procedures / Functions.

## CREATE INDEX

```
CREATE [UNIQUE] INDEX <index> ON <table> (<column> [DESC] [, ...]) [DESC];
```

Creates an index on one or more columns in a table.

Creating an index on searched columns may improve performance. The qualifier DESC can be present for command compatibility with other databases but it has no effect. Unique indexes can be defined but this is deprecated. Use UNIQUE constraints instead. The name of an index must be unique within the whole database.

See also: CREATE TABLE, DROP INDEX.

## CREATE SEQUENCE<sup>1</sup>

```
CREATE SEQUENCE <sequencename> [AS {INTEGER | BIGINT}] [START WITH <startvalue>] [
```

Creates a sequence. The default type is INTEGER. The default start value is 0 and the increment 1. Negative values are not allowed. If a sequence goes beyond Integer.MAXVALUE or Long.MAXVALUE, the next result is determined by 2's complement arithmetic.

The next value for a sequence can be included in SELECT, INSERT and UPDATE statements as in the following example:

```
SELECT [...,]NEXT VALUE FOR <sequencename> [, ...] FROM <tablename>
```

In the proposed SQL 200n and in the current version, there is no way of retrieving the last returned value of a sequence.

## CREATE TABLE

```
CREATE [MEMORY | CACHED | TEMP1 | TEXT1] TABLE <name>  
( <columnDefinition> [, ...] [, <constraintDefinition>...] );
```

Creates a tables in memory (default) or on disk and only cached in memory. If the database is all-in-memory, both MEMORY and CACHED forms of CREATE TABLE return a MEMORY table. If the database is file based, then MEMORY table contents are persisted to disk.

## Components of a CREATE TABLE command

### columnDefinition

```
columnname Datatype [(columnSize[,precision])] [{DEFAULT
GENERATED BY DEFAULT AS IDENTITY (START WITH <n>[, INCREMENT BY <n>]}]
[[NOT] NULL] [IDENTITY] [PRIMARY KEY]
```

Default values that are allowed are constant values or certain SQL datetime functions.

### Allowed Default Values in Column Definitions

- For character column, a single-quoted string or NULL. The only SQL function that can be used is CURRENT\_USER.
- For datetime columns, a single-quoted DATE, TIME or TIMESTAMP value or NULL. Or a datetime SQL function such as CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP, TODAY, NOW. Each function is allowed for a certain datetime type.
- For BOOLEAN columns, the literals FALSE, TRUE, NULL.
- For numeric columns, any valid number or NULL.
- For binary columns, any valid hex string or NULL.

Only one identity column is allowed in each table. Identity columns are autoincrement columns. They must be of INTEGER or BIGINT type and are automatically primary key columns (as a result, multi-column primary keys are not possible with an IDENTITY column present). Using the long SQL syntax the (START WITH <n>) clause specifies the first value that will be used. The last inserted value into an identity column for a connection is available using the function IDENTITY(), for example (where Id is the identity column):

```
INSERT INTO Test (Id, Name) VALUES (NULL, 'Test');
CALL IDENTITY();
```

### constraintDefinition

```
[CONSTRAINT <name>]
UNIQUE ( <column> [,<column>...] ) |
PRIMARY KEY ( <column> [,<column>...] ) |
FOREIGN KEY ( <column> [,<column>...] ) REFERENCES <table> (<column>)
[ON {DELETE | UPDATE} {CASCADE | SET DEFAULT | SET NULL}]
CHECK(<search condition>)1
```

### search condition

A search condition is similar to the set of conditions in a WHERE clause. In the current version of HSQLDB, the conditions for a

CHECK constraint can only reference the current row, meaning there should be no SELECT statement. Sample table definitions with CHECK constraints are in `TestSelfCheckConstraints.txt`. This file is in the `/hsqldb/testrun/hsqldb/` directory of the zip.

#### General syntax limitations

Any supplied column size and precision for numeric types is not enforced and serves only for command compatibility with other databases. With character types (CHARACTER and VARCHAR) these are enforced only if the database property, `sql.enforce_strict_size` has been set to true.

NOT NULL constraints can be part of the column definition only. Other constraints cannot be part of the column definition and must appear at the end of the column definition list.

See also: DROP TABLE.

## CREATE TRIGGER<sup>1</sup>

```
CREATE TRIGGER <name> {BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON <table> [FOR
```

TriggerClass is an application-supplied class that implements the `org.hsqldb.Trigger` interface e.g. "mypackage.TrigClass". It is the fire method of this class that is invoked when the trigger event occurs. You should provide this class, which can have any name, and ensure that this TriggerClass is present in the classpath which you use to start hsqldb.

In 1.7.2 the implementation has been changed and enhanced. When the 'fire' method is called, it is passed the following arguments:

```
fire (String name, String table, Object row1[], Object row2[])
```

where 'row1' and 'row2' represent the 'before' and 'after' states of the row acted on, with each column being a member of the array. The mapping of members of the row arrays to database types is specified in Data Types. For example, BIGINT is represented by a `java.lang.Long` Object. Note that the number of elements in the row arrays is larger than the number of columns by one or two elements. Never modify the last elements of the array, which are not part of the actual row.

If the trigger method wants to access the database, it must establish its own JDBC connection. This can cause data inconsistency and other problems so it is not recommended. The `jdbc:default:connection:URL` is not currently supported.

#### Implementation note:

If QUEUE 0 is specified, the fire method is executed in the same thread as the database engine. This allows trigger action to alter the data that is about to be stored in the database. Data can be checked or modified in BEFORE INSERT / UPDATE + FOR EACH ROW triggers. All table constraints are then enforced by the database engine and if there is a violation, the action is rejected for the SQL command that initiated the INSERT or UPDATE. There is an exception to this rule, that is with UPDATE queries, referential integrity and cascading actions resulting from ON UPDATE CASCADE / SET NULL / SET DEFAULT are all performed prior to the invocation of the trigger method. If an invalid value that breaks referential integrity is inserted in the row by the trigger method, this action is not checked and results in

inconsistent data in the table.

Alternatively, if the trigger is used for external communications and not for checking or altering the data, a queue size larger than zero can be specified. This is in the interests of not blocking the database's main thread as each trigger will run in a thread that will wait for its firing event to occur. When this happens, the trigger's thread calls `TriggerClass.fire`. There is a queue of events waiting to be run by each trigger thread. This is particularly useful for 'FOR EACH ROW' triggers, when a large number of trigger events occur in rapid succession, without the trigger thread getting a chance to run. If the queue becomes full, subsequent additions to it cause the database engine to suspend awaiting space in the queue. Take great care to avoid this situation if the trigger action involves accessing the database, as deadlock will occur. This can be avoided either by ensuring the `QUEUE` parameter makes a large enough queue, or by using the `NOWAIT` parameter, which causes a new trigger event to overwrite the most recent event in the queue. The default queue size is 1024. Note also that the timing of trigger method calls is not guaranteed, so applications should implement their own synchronization measures if necessary.

With a non-zero `QUEUE` parameter, if the trigger methods modifies the 'row2' values, these changes may or may not affect the database and will almost certainly result in data inconsistency.

Please refer to the code for `org.hsquidb.sample.Trigger` [[../src/org/hsquidb/Trigger.html](#)] and `org.hsquidb.sample.TriggerSample` [[../src/org/hsquidb/sample/TriggerSample.html](#)] for more information on how to write a trigger class.

See also: `DROP TRIGGER`.

## CREATE USER

```
CREATE USER username PASSWORD password [ADMIN];
```

Creates a new user or new administrator in this database. Empty password can be made using `''`. You can change a password afterward using an `ALTER USER`<sup>1</sup> command.

Only an administrator can do this.

See also: `CONNECT`, `GRANT`, `REVOKE`. `ALTER USER`<sup>1</sup>,

## CREATE VIEW<sup>1</sup>

```
CREATE VIEW <viewname>[(<viewcolumn>,...) AS SELECT ... FROM ... [WHERE Expression]
```

A view can be thought of as either a virtual table or a stored query. The data accessible through a view is not stored in the database as a distinct object. What is stored in the database is a `SELECT` statement. The result set of the `SELECT` statement forms the virtual table returned by the view. A user can use this virtual table by referencing the view name in `SQL` statements the same way a table is referenced. A view is used to do any or all of these functions:

- Restrict a user to specific rows in a table. For example, allow an employee to see only the rows recording his or her work in a labor-tracking table.
- Restrict a user to specific columns. For example, allow employees who do not work in payroll to see the name, office, work phone, and department columns in an employee table, but do not allow them to see any columns with salary information or personal information.
- Join columns from multiple tables so that they look like a single table.

- Aggregate information instead of supplying details. For example, present the sum of a column, or the maximum or minimum value from a column.

Views are created by defining the SELECT statement that retrieves the data to be presented by the view. The data tables referenced by the SELECT statement are known as the base tables for the view. In this example, is a view that selects data from three base tables to present a virtual table of commonly needed data:

```
CREATE VIEW mealsjv AS
  SELECT m.mid mid, m.name name, t.mealtype mt, a.aid aid,
         a.gname + ' ' + a.sname author, m.description description,
         m.asof asof
  FROM meals m, mealtypes t, authors a
  WHERE m.mealtype = t.mealtype
  AND m.aid = a.aid;
```

You can then reference mealsjv in statements in the same way you would reference a table:

```
SELECT * FROM mealsjv;
```

A view can reference another view. For example, mealsjv presents information that is useful for long descriptions that contain identifiers, but a short list might be all a web page display needs. A view can be built that selects only specific mealsjv columns:

```
CREATE VIEW mealswebv AS SELECT name, author FROM mealsjv;
```

The SELECT statement in a VIEW definition should return columns with distinct names. If the names of two columns in the SELECT statement are the same, use a column alias to distinguish between them. A list of new column names can always be defined for a view.

```
CREATE VIEW aview (new_name, new_author) AS
  SELECT name, author
  FROM mealsjv
```

See also: SQL Expression, SELECT<sup>1</sup>, DROP VIEW<sup>1</sup>.

## DELETE

```
DELETE FROM table [WHERE Expression];
```

Removes rows in a table.

See also: SQL Expression, INSERT, SELECT<sup>1</sup>.

## DISCONNECT

```
DISCONNECT;
```

Closes this connection. It is not required to call this command when using the JDBC interface: it is called automatically when the connection is closed. After disconnecting, it is not possible to execute other queries (including CONNECT) with this connection.

See also: CONNECT.

## DROP INDEX

```
DROP INDEX index [IF EXISTS];
```

Removes the specified index from the database. Will not work if the index backs a UNIQUE or FOREIGN KEY constraint.

See also: CREATE INDEX.

## DROP SEQUENCE<sup>1</sup>

```
DROP SEQUENCE <sequencename>;
```

Removes the specified sequence from the database.

## DROP TABLE

```
DROP TABLE <table> [IF EXISTS];
```

Removes a table, the data and indexes from the database. When IF EXIST is used, the statement returns without an error even if the table does not exist. Will fail if the table has been reference by a foreign key or a view.

See also: CREATE TABLE.

## DROP TRIGGER

```
DROP TRIGGER <trigger>;
```

Removes a trigger from the database.

See also: CREATE TRIGGER<sup>1</sup>.

## DROP USER

```
DROP USER <username>;
```

Removes a user from the database.

Only an administrator do this.



See also: CREATE USER.

## DROP VIEW<sup>1</sup>

```
DROP VIEW <viewname> [IF EXISTS];
```

Removes a view from the database. When IF EXIST is used, the statement returns without an error if the view does not exist.

See also: CREATE VIEW<sup>1</sup>.

## GRANT

```
GRANT { SELECT | DELETE | INSERT | UPDATE | ALL } [,...]
ON { table | CLASS "package.class" } TO { username | PUBLIC };
```

Assigns privileges to a user or to all users (PUBLIC) for a table or for a class. To allow a user to call a function from a class, the right ALL must be used. Examples:

```
GRANT SELECT ON Test TO GUEST;
GRANT ALL ON CLASS "java.lang.String" TO PUBLIC;
```

Only an administrator do this.

See also: REVOKE, CREATE USER.

## INSERT

```
INSERT INTO table [( column [,...] )]
{ VALUES(Expression [,...]) | SelectStatement};
```

Adds one or more new rows of data into a table.

## REVOKE

```
REVOKE { SELECT | DELETE | INSERT | UPDATE | ALL } [,...]
ON { table | CLASS "package.class" } TO { username | PUBLIC };
```

Withdraws privileges from a user or for PUBLIC (all users) for a table or class.

Only an administrator may do this.

See also: GRANT.

## ROLLBACK

```
ROLLBACK [TO SAVEPOINT <savepoint name>1 | WORK];
```

ROLLBACK used on its own, or with WORK, undoes changes made since the last COMMIT or ROLLBACK.

ROLLBACK TO SAVEPOINT <savepoint name> undoes the change since the named savepoint. It has no effect if the savepoint is not found.

See also: COMMIT.

## SAVEPOINT<sup>1</sup>

```
SAVEPOINT <savepoint name>;
```

Sets up a SAVEPOINT for use with ROLLBACK TO SAVEPOINT.

See also: COMMIT.

## SCRIPT

```
SCRIPT ['file'];
```

Creates an SQL script describing the database. If the file is not specified, a result set containing only the DDL script is returned. If the file is specified then this file is saved with the path relative to the machine where the database engine is located.

Only an administrator may do this.

## SELECT<sup>1</sup>

```
SELECT [{LIMIT n m | TOP m}1][ALL | DISTINCT]
{ selectExpression | table.* | * } [, ...]
[INTO [CACHE | TEMP | TEXT]1 newTable]
FROM tableList
[WHERE Expression]
[GROUP BY Expression [, ...]]
[HAVING Expression]
[ { UNION [ALL | DISTINCT] | {MINUS [DISTINCT] | EXCEPT [DISTINCT]} | INTERSECT [DISTINCT] }
[ORDER BY orderExpression [, ...]]];
```

Retrieves information from one or more tables in the database.

### Components of a SELECT command

tableList

```
table [{ INNER | LEFT OUTER } JOIN table ON Expression]
```

table

```
{ (selectStatement) [AS] label | tableName }
```

selectExpression

```
{ Expression | COUNT(*) | {COUNT | MIN | MAX | SUM |
```

If **DISTINCT** is specified, only one instance of several equivalent values is used in the aggregate function.

orderExpression

```
{ columnNr | columnAlias | selectExpression } [ASC |
```

**LIMIT** n m

Creates the result set for the **SELECT** statement first and then discards the first n rows and returns the first m rows of the remaining result set. Special cases: **LIMIT** 0 m is equivalent to **TOP** m or **FIRST** m in other RDBMS's; **LIMIT** n 0 discards the first n rows and returns the rest of the result set.

**TOP** m

Equivalent to **LIMIT** 0 m.

**UNION** and other set operations

Multiple **SELECT** statements joined with **UNION**, **EXCEPT** and **INTERSECT** are possible. Each **SELECT** is then treated as a term, and the set operation as an operator in an expression. The expression is evaluated from left to right but **INTERSECT** takes precedence over the rest of the operators and is applied first. You can use parentheses around any number of **SELECT** statements to change the evaluation order.

**NOT NULL** constraints can be part of the column definition only. Other constraints cannot be part of the column definition and must appear at the end of the column definition list.

See also: **INSERT**, **UPDATE**, **DELETE**.

## SET AUTOCOMMIT

```
SET AUTOCOMMIT { TRUE | FALSE };
```

Switches on or off the connection's auto-commit mode. If switched on, then all statements will be committed as individual transactions. Otherwise, the statements are grouped into transactions that are terminated by either **COMMIT** or **ROLLBACK**. By default, new connections are in auto-commit mode. This command should not be used directly. Use the JDBC equivalent method, `Connection.setAutoCommit(boolean autocommit)`.

## SET IGNORECASE

```
SET IGNORECASE { TRUE | FALSE };
```

Disables (`ignorecase = true`) or enables (`ignorecase = false`) the case sensitivity of text comparison and indexing for new tables. By default, character columns in new databases are case sensitive. The sensitivity must be switched before creating tables. Existing tables and their data are not affected. When switched on, the data type `VARCHAR` is set to `VARCHAR_IGNORECASE` in new tables. Alternatively, you can 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.

Only an administrator may do this.

## SET LOGSIZE

```
SET LOGSIZE size;
```

Sets the maximum size in MB of the `.log` file. Default is 200 MB. The database will be closed and opened (just like using `CHECKPOINT`) if the `.log` file gets over this limit, and so the `.log` file will shrink. 0 means no limit.

See also: `CHECKPOINT`.

## SET PASSWORD

```
SET PASSWORD password;
```

Changes the password of the currently connected user. Empty password can be set using `''`.

## SET PROPERTY<sup>1</sup>

```
SET PROPERTY <double quoted name> <value>;
```

Sets a database property. Properties that can be set using this command are either boolean or integral and are listed in the `Advanced Topics` chapter.

## SET REFERENTIAL INTEGRITY

```
SET REFERENTIAL_INTEGRITY { TRUE | FALSE };
```

This command enables / disables the referential integrity checking (foreign keys). Normally it should be switched on (this is the default) but when importing data (and the data is imported in the 'wrong' order) the checking can be switched off.

### Warning

Note that when referential integrity is switched back on, no check is made that the changes to the data are consistent with the existing referential integrity constraints.

Only an administrator may do this.

See also: `CREATE TABLE`.

## SET SCRIPTFORMAT<sup>1</sup>

```
SET SCRIPTFORMAT {TEXT | BINARY | COMPRESSED};
```

Changes the format of the script file. BINARY and COMPRESSED formats are slightly faster and more compact than the default TEXT. Recommended only for very large script files.

## SET TABLE INDEX

```
SET TABLE tableName INDEX 'index1rootPos index2rootPos ... ';
```

This command is only used internally to store the position of index roots in the .data file. It appears only in database script files; it should not be used directly.

## SET TABLE READONLY<sup>1</sup>

```
SET TABLE <tablename> READONLY {TRUE | FALSE};
```

Sets the table as read only.

## SET TABLE SOURCE<sup>1</sup>

```
SET TABLE <tablename> SOURCE <file and options> [DESC];
```

For details see the Text Tables chapter.

This command is used exclusively with TEXT tables to specify which file is used for storage of the data. The optional DESC qualifier results in the text file indexed from the end and opened as readonly. The <file and options> argument is a double quoted string that consists of:

```
<file and options> ::= <doublequote> <filepath> [<semicolon> <option>...] <doub
```

Example:

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

### Special indicators for Hsqldb Text Table separators

\semi	semicolon
\quote	quote
\space	space character
\apos	apostrophe

<code>\n</code>	newline - Used as an end anchor (like \$ in regular expressions)
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\\</code>	backslash
<code>\u####</code>	a Unicode character specified in hexadecimal

Only an administrator may do this.

## SET WRITE DELAY

```
SET WRITE_DELAY { { TRUE | FALSE } | <seconds> };
```

In 1.7.2 this controls the frequency of file synch. When `WRITE_DELAY` is set to `FALSE`, the synch takes place once every second. `WRITE_DELAY TRUE` performs the synch once every minute. The default is `TRUE` (60 seconds). A numeric value can be specified instead.

Only an administrator may do this.

## SHUTDOWN

```
SHUTDOWN [ IMMEDIATELY | COMPACT | SCRIPT1 ];
```

Closes the current database.

### Varieties of the SHUTDOWN command

Only an administrator may use the `SHUTDOWN` command.

<code>SHUTDOWN</code>	Performs a checkpoint to creates a new <code>.script</code> file that has the minimum size and contains the data for memory tables only. It then backs up the <code>.data</code> file containing the <code>CACHED TABLE</code> data in zipped format to the <code>.backup</code> file and closes the database.
<code>SHUTDOWN IMMEDIATELY</code>	Just closes the database files (like when the Java process for the database is terminated); this command is used in tests of the recovery mechanism. This command should not be used as the routine method of closing the database.
<code>SHUTDOWN COMPACT</code>	Writes out a new <code>.script</code> file which contains the data for all the tables, including <code>CACHED</code> and <code>TEXT</code> tables. It then deletes the existing text table files and the <code>.data</code> file before rewriting them. After this, it backs up the <code>.data</code> file in the same way as normal <code>SHUTDOWN</code> . This operation shrinks all files to the minimum size.
<code>SHUTDOWN SCRIPT</code>	Similar to <code>SHUTDOWN COMPACT</code> but after writing the script and deleting the existing files, it does not rewrite the <code>.data</code> and text table files. After <code>SHUTDOWN SCRIPT</code> , only the <code>.script</code> and <code>.properties</code> file re-

main. 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.

## UPDATE

```
UPDATE table SET column = Expression [, ...] [WHERE Expression];
```

Modifies data of a table in the database.

See also: SELECT<sup>1</sup>, INSERT, DELETE.

## Data Types

**Table 8.1. Data Types. The types on the same line are equivalent.**

Name	Range	Java Type
INTEGER   INT	as Java type	int   java.lang.Integer
DOUBLE [PRECISION]   FLOAT	as Java type	double   java.lang.Double
VARCHAR	as Integer.MAXVALUE	java.lang.String
VARCHAR_IGNORECASE	as Integer.MAXVALUE	java.lang.String
CHAR   CHARACTER	as Integer.MAXVALUE	java.lang.String
LONGVARCHAR	as Integer.MAXVALUE	java.lang.String
DATE	as Java type	java.sql.Date
TIME	as Java type	java.sql.Time
TIMESTAMP   DATETIME	as Java type	java.sql.Timestamp
DECIMAL	No limit	java.math.BigDecimal
NUMERIC	No limit	java.math.BigDecimal
BOOLEAN   BIT	as Java type	boolean   java.lang.Boolean
TINYINT	as Java type	byte   java.lang.Byte
SMALLINT	as Java type	short   java.lang.Short
BIGINT	as Java type	long   java.lang.Long
REAL	as Java type	double   java.lang.Double <sup>1</sup>
BINARY	as Integer.MAXVALUE	byte[]
VARBINARY	as Integer.MAXVALUE	byte[]
LONGVARBINARY	as Integer.MAXVALUE	byte[]
OTHER   OBJECT	as Integer.MAXVALUE	java.lang.Object

The uppercase names are the data types names defined by the SQL standard or commonly used by RDMS's. The data types in quotes are the Java class names - if these type names are used then they must be enclosed in quotes because in Java names are case-sensitive. Range indicates the maximum size of the object that can be stored. Where Integer.MAXVALUE is stated, this is a theoretical limit and in practice the maximum size of a VARCHAR or BINARY object that can be stored is dictated by the amount of memory available. In practice, objects of up to a megabyte in size have been successfully used in production databases.

The recommended Java mapping for the JDBC datatype FLOAT is as a Java type "double". Because of the potential confusion it is recommended that DOUBLE is used instead of FLOAT.

VARCHAR\_IGNORECASE is a special case-insensitive type of VARCHAR. This type is not portable.

In table definition statements, HSQLDB accepts size, precision and scale qualifiers only for certain types: CHAR(n), VARCHAR(n), DOUBLE(n), DECIMAL(p,s).

The specified precision and scale for DOUBLE and DECIMAL is simply ignored by the engine. Instead, the values for the corresponding Java types are always used, which in the case of DECIMAL is an unlimited precision and scale. When defining CHAR and VARCHAR columns, the SIZE argument is optional and defaults to 0. If any other size is specified, it is stored in the database definition but is not enforced by default. Once you have created the database (before adding data), you can add a database property value to enforce the sizes:

```
SET PROPERTY "sql.enforce_strict_size" true
```

This will enforce the specified size and pad CHAR fields with spaces to fill the size. This complies with SQL standards by throwing an exception if an attempt is made to insert a string longer than the maximum size.

CHAR and VARCHAR and LONGVARCHAR columns are by default compared and sorted according to POSIX standards. To use the current JRE locale for sorting and comparison, add the following database property to the properties file.

```
sql.compare_in_locale=true
```

Columns of the type OTHER or OBJECT contain the serialized form of a Java Object in binary format. To insert or update such columns, a binary format string (see below under Expression) should be used. Using PreparedStatement with JDBC automates this transformation.

## SQL Comments

```
-- SQL style line comment  
// Java style line comment  
/* C style line comment */
```

All these types of comments are ignored by the database.

## Stored Procedures / Functions

Stored procedures are Java functions that are called directly from the SQL language or using an alias. Calling Java functions (directly or using the alias) requires that the Java class can be reached by the database (server). The syntax is:



```
"java.lang.Math.sqrt"(2.0)
```

This means the package must be provided, and the name must be written as one word, and inside " because otherwise it is converted to uppercase (and not found).

An alias can be created using the command CREATE ALIAS:

```
CREATE ALIAS SQRT FOR "java.lang.Math.sqrt";
```

When an alias is defined, then the function can be called additionally using this alias:

```
SELECT SQRT(A) , B FROM MYTABLE;
```

Only static java methods can be used as stored procedures. If, within the same class, there are over-loaded methods with the same number of arguments, then the first one encountered by the program will be used. If you want to use Java library methods, it is recommended that you create your own class with static methods that act as wrappers around the Java library methods. This will allow you to control which method signature is used to call each Java library method.

## Built-in Functions and Stored Procedures

### Numerical built-in Functions / Stored Procedures

ABS(d)	returns the absolute value of a double value
ACOS(d)	returns the arc cosine of an angle
ASIN(d)	returns the arc sine of an angle
ATAN(d)	returns the arc tangent of an angle
ATAN2(a,b)	returns the tangent of a/b
BITAND(a,b)	return a & b
BITOR(a,b)	returns a   b
CEILING(d)	returns the smallest integer that is not less than d
COS(d)	returns the cosine of an angle
COT(d)	returns the cotangent of an angle
DEGREES(d)	converts radians to degrees
EXP(d)	returns e (2.718...) raised to the power of d
FLOOR(d)	returns the largest integer that is not greater than d
LOG(d)	returns the natural logarithm (base e)

LOG10(d)	returns the logarithm (base 10)
MOD(a,b)	returns a modulo b
PI()	returns pi (3.1415...)
POWER(a,b)	returns a raised to the power of b
RADIANS(d)	converts degrees to radians
RAND()	returns a random number x bigger or equal to 0.0 and smaller than 1.0
ROUND(a,b)	rounds a to b digits after the decimal point
ROUNDMAGIC(d)	solves rounding problems such as 3.11-3.1-0.01
SIGN(d)	returns -1 if d is smaller than 0, 0 if d==0 and 1 if d is bigger than 0
SIN(d)	returns the sine of an angle
SQRT(d)	returns the square root
TAN(A)	returns the trigonometric tangent of an angle
TRUNCATE(a,b)	truncates a to b digits after the decimal point

### String built-in Functions / Stored Procedures

ASCII(s)	returns the ASCII code of the leftmost character of s
BIT_LENGTH(str) <sup>1</sup>	returns the length of the string in bits
CHAR(c)	returns a character that has the ASCII code c
CHAR_LENGTH(str) <sup>1</sup>	returns the length of the string in characters
CONCAT(str1,str2)	returns str1 + str2
DIFFERENCE(s1,s2)	returns the difference between the sound of s1 and s2
HEXTORAW(s1) <sup>1</sup>	returns translated string
INSERT(s,start,len,s2)	returns a string where len number of characters beginning at start has been replaced by s2
LCASE(s)	converts s to lower case
LEFT(s,count)	returns the leftmost count of characters of s) - requires double quoting - use SUBSTRING() instead
LENGTH(s)	returns the number of characters in s
LOCATE(search,s,[start])	returns the first index (1=left, 0=not found) where search is found in s, starting at start
LTRIM(s)	removes all leading blanks in s

OCTET_LENGTH(str) <sup>1</sup>	returns the length of the string in bytes (twice the number of characters)
RAWTOHEX(s1) <sup>1</sup>	returns translated string
REPEAT(s,count)	returns s repeated count times
REPLACE(s,replace,s2)	replaces all occurrences of replace in s with s2
RIGHT(s,count)	returns the rightmost count of characters of s
RTRIM(s)	removes all trailing spaces
SOUNDEX(s)	returns a four character code representing the sound of s
SPACE(count)	returns a string consisting of count spaces
SUBSTR(s,start[,len])	alias for substring
SUBSTRING(s,start[,len])	returns the substring starting at start (1=left) with length len
UCASE(s)	converts s to upper case
LOWER(s)	converts s to lower case
UPPER(s)	converts s to upper case

### **Date/Time built-in Functions / Stored Procedures**

CURDATE()	returns the current date
CURTIME()	returns the current time
DATEDIFF(string, datetime1, datetime2) <sup>1</sup>	returns the count of units of time elapsed from datetime1 to datetime2. The string indicates the unit of time and can have the following values 'ms'='millisecond', 'ss'='second', 'mi'='minute', 'hh'='hour', 'dd'='day', 'mm'='month', 'yy' = 'year'. Both the long and short form of the strings can be used.
DAYNAME(date)	returns the name of the day
DAYOFMONTH(date)	returns the day of the month (1-31)
DAYOFWEEK(date)	returns the day of the week (1 means Sunday)
DAYOFYEAR(date)	returns the day of the year (1-366)
HOUR(time)	return the hour (0-23)
MINUTE(time)	returns the minute (0-59)
MONTH(date)	returns the month (1-12)
MONTHNAME(date)	returns the name of the month
NOW()	returns the current date and time as a timestamp) - use CURRENT_TIMESTAMP instead

QUARTER(date)	returns the quarter (1-4)
SECOND(time)	returns the second (0-59)
WEEK(date)	returns the week of this year (1-53)
YEAR(date)	returns the year
CURRENT_DATE <sup>1</sup>	returns the current date
CURRENT_TIME <sup>1</sup>	returns the current time
CURRENT_TIMESTAMP <sup>1</sup>	returns the current timestamp

### **System/Connection built-in Functions / Stored Procedures**

DATABASE()	returns the name of the database of this connection
USER()	returns the user name of this connection
CURRENT_USER	SQL standard function, returns the user name of this connection
IDENTITY()	returns the last identity values that was inserted by this connection

### **System built-in Functions / Stored Procedures**

IFNULL(exp,value)	if exp is null, value is returned else exp) - use COALESCE() instead
CASEWHEN(exp,v1,v2)	if exp is true, v1 is returned, else v2) - use CASE WHEN instead
CONVERT(term,type)	converts exp to another data type
CAST(term AS type)	converts exp to another data type

1	if expr1 is not null then it is returned else, expr2 is evaluated and if not null it is returned and so on
NULLIF(v1,v2) <sup>1</sup>	if v1 equals v2 return null, otherwise v1
CASE v1 WHEN... <sup>1</sup>	CASE v1 WHEN v2 THEN v3 [ELSE v4] END  when v1 equals v2 return v3 [otherwise v4 or null if there is no ELSE]
CASE WHEN... <sup>1</sup>	CASE WHEN expr1 THEN v1[WHEN expr2 THEN v2] [ELSE v4] END  when expr1 is true return v1 [optionally repeated for more cases] [otherwise v4 or null if there is no ELSE]
EXTRACT <sup>1</sup>	EXTRACT ({YEAR   MONTH   DAY   HOUR   MINUTE   SECOND} FROM <datetime value>)
POSITION ... IN <sup>1</sup>	POSITION(<string expression> IN <string expression>)  if the first string is a sub-string of the second one, returns the position of the sub-string, counting from one; otherwise 0
SUBSTRING... FROM <sup>1</sup>	SUBSTRING(<string expression> FROM <numeric expression> [FOR <numeric expression>])
TRIM... FROM <sup>1</sup>	TRIM([LEADING   TRAILING   BOTH]) FROM <string expression>)

See also: CALL, CREATE ALIAS.

## SQL Expression

[NOT] condition [{ OR | AND } condition]

### Components of SQL Expressions

condition

```
{ value [| value]
  value { = | < | <= | > | >= | <> | != | IS [NOT] } value
  EXISTS(selectStatement)
  value BETWEEN value AND value
  value [NOT] IN ( {value [, ...] | selectStatement } )
  value [NOT] LIKE value [ESCAPE] value }
```

value

```
[+ | -] { term [{ + | - | * | / | || } term]
  ( condition )
  function ( [parameter] [,...] )
  selectStatement giving one value
```

term

```
{ 'string' | number | floatingpoint
  | [table.]column | TRUE | FALSE | NULL }
```

sequence

```
NEXT VALUE FOR <sequence>
```

HSQLDB does not currently enforce the SQL 200n proposed rules on where sequence generated values are allowed to be used. In general, these values can be used in insert and update statements but not in CASE statements, order by clauses, search conditions, aggregate functions, or grouped queries.

string

Strings in HSQLDB are Unicode strings. A string starts and ends with a single ' (singlequote). In a string started with ' (singlequote) use " (two singlequotes) to create a ' (singlequote).

String contatenation should be performed with the standard SQL operator || rather than the non-standard + operator.

The LIKE keyword uses '%' to match any (including 0) number of characters, and '\_' to match exactly one character. To search for '%' or '\_' itself an escape character must also be specified using the ESCAPE clause. For example, if the backslash is the escaping character, '%' and '\_' can be used to find the '%' and '\_' characters themselves. For example, `SELECT .... LIKE '\_%' ESCAPE '\' will find the strings beginning with an underscore.`

name

The character set for quoted identifiers (names) in HSQLDB is Unicode.

A unquoted identifier (name) starts with a letter and is followed by any number of ASCII letters or digits. When an SQL statement is issued, any lowercase characters in unquoted identifiers are converted to uppercase. Because of this, unquoted names are in fact ALL UPPERCASE when used in SQL statements. An important implication of this is the for accessing columns names via JDBC DatabaseMetaData: the internal form, which is the ALL UPPERCASE must be used if the column name was not quoted in the CREATE TABLE statement.

Quoted identifiers can be used as names (for tables, columns, constraints or indexes). Quoted identifiers start and end with " (one doublequote). A quoted identifier can contain any Unicode character, including space. In a quoted identifier use "" (two doublequotes) to create a " (one doublequote). With quoted identifiers it is possible to create mixed-case table and column names. Example:

```
CREATE TABLE "Address" ("Nr" INTEGER, "Name" VARCHAR); SELECT "Nr" ,
```

The equivalent quoted identifier can be used for an unquoted identifier by converting the identifier to all uppercase and quoting it. For example, if a table name is defined as Address2 (unquoted), it can be referred to by its quoted form, "ADDRESS2", as well as address2, aDdress2 and ADDRESS2. Quoted identifiers should not be confused with SQL strings.

Quoting can sometimes be used for identifiers, aliases or functions when there is an ambiguity. For example:

```
SELECT COUNT(*) "COUNT" FROM MYTABLE;  
SELECT "LEFT"(COL1, 2) FROM MYTABLE;
```

Although HSQLDB 1.7.2 does not force unquoted identifiers to contain only ASCII characters, the use of non-ASCII characters in these identifiers does not comply with SQL standards. Portability between different JRE locales could be an issue when accented characters (or extended unicode characters) are used in unquoted identifiers. Because native Java methods are used to convert the identifier to uppercase, the result may vary not be expected in different locales. It is recommended that accented characters are used only in quoted identifiers.

When using JDBC DatabaseMetaData methods that take table, column, or index identifiers as arguments, treat the names as they are registered in the database. With these methods, unquoted identifiers should be used in all-uppercase to get the correct result. Quoted identifiers should be used in the exact case combination as they were defined - no quote character should be included around the name. JDBC methods that return a result set containing such identifiers return unquoted identifiers as all-uppercase and quoted identifiers in the exact case they are registered in the database (a change from 1.6.1 and previous versions).

values

- A DATE literal starts and ends with ' (singlequote), the format is yyyy-mm-dd (see `java.sql.Date`).
- A TIME literal starts and ends with ' (singlequote), the format is hh:mm:ss (see `java.sql.Time`).
- A TIMESTAMP or DATETIME literal starts and ends with ' (singlequote), the format is yyyy-mm-dd hh:mm:ss.SSSSSSSS (see `java.sql.Timestamp`).

When specifying default values for date / time columns in CREATE TABLE statements, or in SELECT, INSERT, and UPDATE statements, special SQL functions: NOW, SYS-DATE, TODAY, CURRENT\_TIMESTAMP, CURRENT\_TIME and CURRENT\_DATE (case independent) can be used. NOW is used for TIME and TIMESTAMP columns, TODAY is used for DATE columns. The data and time variants CURRENT\_\* are SQL standard versions and should be used in preference to others. Example:

```
CREATE TABLE T(D DATE DEFAULT CURRENT_DATE);  
CREATE TABLE T1(TS TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

Binary data starts and ends with ' (singlequote), the format is hexadecimal. '0004ff' for example is 3 bytes, first 0, second 4 and last 255 (0xff).

Any number of commands may be combined. With combined commands, ';' (semicolon) must be used at the end of each command to ensure data integrity, despite the fact that the engine may understand the end of commands and not return an error when a semicolon is not used.

---

# Appendix A. Building Hsqldb version 1.7.2

Fred Toussi, HSQLDB Development Group <ft@cluedup.com>  
\$Date: 2004/06/06 08:45:56 \$

## Purpose

From 1.7.2, the supplied `hsqldb.jar` file is built with Java 1.4.2. If you want to run the engine under JDK1.3 or earlier, you should rebuild the jar with Ant.

## Building with Ant, from the Apache Jakarta Project

Ant (Another Neat Tool) is used for building hsqldb. The version currently used to test the build script is 1.6.1 but versions since 1.5.1 should also be compatible.

## 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

<code>jar</code>	to make the <code>hsqldb.jar</code>
<code>clean</code>	to clean up the <code>/classes</code> directory that is created
<code>cleanall</code>	to remove the old jar as well
<code>javadoc</code>	to build javadoc
<code>jarmain</code>	to build a smaller jar for hsqldb that does not contain utilities
<code>jarclient</code>	to build an extremely small jar containing only the client-side JDBC driver (does not support direct connection to HSQLDB URLs of the form <code>jdbc:hsldb:mem:*</code> , <code>jd-</code>



```
bc:hsqldb:file:*, nor jdbc:hsqldb:res:*)  
jarmin      to build a very small jar with no support for servers  
jartest     to build a larger jar for hsqldb that contains tests  
...        Many more targets are available
```

HSQldb can be built in any combination of five different sizes and three JRE (Java Runtime Environment) versions. The smallest jar size (`hsqldbjdbc.jar`) contains only the HSQldb JDBC Driver client. The next smallest jar size (`hsqldbmin.jar`) contains only the standalone database (no servers) and JDBC support and is suitable for embedded applications. The default size (`hsqldb.jar`) also contains server mode support and the utilities. The largest size (`hsqldbtest.jar`) includes some test classes as well. (You can also build `hsqldbtool.jar`. If you use `SqlTool`, see the `SqlTool` chapter about that.)

In order to build and run the test classes, you need the JUnit jar in the `/lib` directory. This is available from <http://www.junit.org>.

The preferred method of rebuilding the jar is with Ant. After installing Ant on your system use the following command from the `/build` directory:

```
ant explainjars
```

The command displays a list of different options for building different sizes of the HSQldb Jar. The default is built using:

### Example A.1. Building the standard Hsqldb jar file with Ant

```
ant jar
```

The Ant method always builds a jar with the JDK that is used by Ant and specified in the `JAVA_HOME` environment variable. Building with JDK 1.4.x will result in a jar that is not backward compatible. From version 1.7.2, use of JDK 1.1.x is not recommended for building the JAR, even for running under JDK 1.1.x -- use JDK 1.3.1 for compatibility with 1.1.x.

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.

## Building with DOS Batch Files

UNIX users must use Ant to build hsqldb.

For DOS/Windows users, a set of MSDOS batch files is provided as an alternative to using Ant. These produce only the default jar size. The path and classpath variables for the JDK should of course be set before running any of the batch files.

If you are compiling for JDK's other than 1.4.x, you should use the appropriate `switchToJDK11.bat` or `switchToJDK12.bat` to adapt the source files to the target JDK before running the appropriate `buildJDK11.bat` or `buildJDK12.bat` JDK and JRE versions.

From version 1.7.2, Use of JDK 1.1.x is not recommended for building the JAR, even for running under JDK 1.1.x -- use JDK1.3 for running under 1.1.x. Javadoc can be built with Ant and batch files.

## Hsqldb CodeSwitcher

CodeSwitcher is a tool to manage different version of Java source code. It allows to compile HSQLDB 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 HSQLDB build scripts. You do not have to use it separately to compile HSQLDB.

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 A.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 A.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 A.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 `switchtojdk1*.bat` files in the `/build` directory.

---

# Appendix B. First JDBC Client Example

There is a copy of Testdb.java in the directory src/org/hsqldb/sample of your HSQLDB distribution.

## Example B.1. JDBC Client source code example

```
/* Copyright (c) 2001-2004, The HSQL Development Group
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * Redistributions of source code must retain the above copyright notice, this
 * list of conditions and the following disclaimer.
 *
 * Redistributions in binary form must reproduce the above copyright notice,
 * this list of conditions and the following disclaimer in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of the HSQL Development Group nor the names of its
 * contributors may be used to endorse or promote products derived from this
 * software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL HSQL DEVELOPMENT GROUP, HSQLDB.ORG,
 * OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

package org.hsqldb.sample;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * Title: Testdb
 * Description: simple hello world db example of a
 * standalone persistent db application
 *
 * every time it runs it adds four more rows to sample_table
 * it does a query and prints the results to standard out
 *
 * Author: Karl Meissner karl@meissnersd.com
 */
public class Testdb {

    Connection conn; //our connect
```

```

// we dont want this garbage collected until we are done
public Testdb(String db_file_name_prefix) throws Exception {    // note more g

    // Load the HSQL Database Engine JDBC driver
    // hsqldb.jar should be in the class path or made part of the current jar
    Class.forName("org.hsqldb.jdbcDriver");

    // connect to the database.  This will load the db files and start the
    // database if it is not already running.
    // db_file_name_prefix is used to open or create files that hold the state
    // of the db.
    // It can contain directory names relative to the
    // current working directory
    conn = DriverManager.getConnection("jdbc:hsqldb:"
                                      + db_file_name_prefix,    // filenames
                                      "sa",                    // username
                                      "");                      // password
}

public void shutdown() throws SQLException {

    conn.close();    // if there are no other open connection

    // db writes out to files and shuts down
    // this happens anyway at garbage collection
    // when program ends
}

//use for SQL command SELECT
public synchronized void query(String expression) throws SQLException {

    Statement st = null;
    ResultSet rs = null;

    st = conn.createStatement();    // statement objects can be reused wi

    // repeated calls to execute but we
    // choose to make a new one each time
    rs = st.executeQuery(expression);    // run the query

    // do something with the result set.
    dump(rs);
    st.close();    // NOTE!! if you close a statement the associated ResultSet

    // closed too
    // so you should copy the contents to some other object.
    // the result set is invalidated also if you recycle an Statement
    // and try to execute some other query before the result set has been
    // completely examined.
}

//use for SQL commands CREATE, DROP, INSERT and UPDATE
public synchronized void update(String expression) throws SQLException {

    Statement st = null;

    st = conn.createStatement();    // statements

    int i = st.executeUpdate(expression);    // run the query

    if (i == -1) {
        System.out.println("db error : " + expression);
    }
}

```

```

        st.close();
    } // void update()

    public static void dump(ResultSet rs) throws SQLException {

        // the order of the rows in a cursor
        // are implementation dependent unless you use the SQL ORDER statement
        ResultSetMetaData meta = rs.getMetaData();
        int colmax = meta.getColumnCount();
        int i;
        Object o = null;

        // the result set is a cursor into the data. You can only
        // point to one row at a time
        // assume we are pointing to BEFORE the first row
        // rs.next() points to next row and returns true
        // or false if there is no next row, which breaks the loop
        for (; rs.next(); ) {
            for (i = 0; i < colmax; ++i) {
                o = rs.getObject(i + 1); // Is SQL the first column is indexed

                // with 1 not 0
                System.out.print(o.toString() + " ");
            }

            System.out.println(" ");
        } //void dump( ResultSet rs )
    }

    public static void main(String[] args) {

        Testdb db = null;

        try {
            db = new Testdb("db_file");
        } catch (Exception ex1) {
            ex1.printStackTrace(); // could not start db

            return; // bye bye
        }

        try {

            //make an empty table
            //
            // by declaring the id column IDENTITY, the db will automatically
            // generate unique values for new rows- useful for row keys
            db.update(
                "CREATE TABLE sample_table ( id INTEGER IDENTITY, str_col VARCHAR(
            ) catch (SQLException ex2) {

                //ignore
                //ex2.printStackTrace(); // second time we run program
                // should throw exception since table
                // already there
                //
                // this will have no effect on the db
            }

        }

        try {

            // add some rows - will create duplicates if run more than once
            // the id column is automatically generated

```

```
db.update(
    "INSERT INTO sample_table(str_col,num_col) VALUES('Ford', 100)");
db.update(
    "INSERT INTO sample_table(str_col,num_col) VALUES('Toyota', 200)");
db.update(
    "INSERT INTO sample_table(str_col,num_col) VALUES('Honda', 300)");
db.update(
    "INSERT INTO sample_table(str_col,num_col) VALUES('GM', 400)");

// do a query
db.query("SELECT * FROM sample_table WHERE num_col < 250");

// at end of program
db.shutdown();
} catch (SQLException ex3) {
    ex3.printStackTrace();
}
} // main()
} // class Testdb
```

---

# Appendix C. Hsqldb Database Files and Recovery

This text is based on HypersonicSQL documentation, updated to reflect the latest version 1.7.2 of HSQLDB.

\$Date: 2004/04/20 18:35:56 \$

The Standalone and Client/Server modes will in most cases use files to store all data to disk in a persistent and safe way. This document describes the meaning of the files, the states and the procedures followed by the engine to recover the data.

A database named 'test' is used in this description. The database files will be `test.script`, `test.properties`, `test.data` and `test.backup`.

## Database Files

<code>test.properties</code>	Contains the entry 'modified'. If the entry 'modified' is set to 'yes' then the database is either running or was not closed correctly (because the close algorithm sets 'modified' to 'no' at the end).
<code>test.script</code>	This file contains the SQL statements that makes up the database up to the last checkpoint - it is in synch with <code>test.backup</code> .
<code>test.data</code>	This file contains the (binary) data records for CACHED tables only.
<code>test.backup</code>	This is compressed file that contains the complete backup of the old <code>test.data</code> file at the time of last checkpoint.
<code>test.log</code>	This file contains the extra SQL statements that have modified the database since the last checkpoint (something like the 'Redo-log' or 'Transaction-log', but just text).

In the above list, a checkpoint results from both a CHECKPOINT command and a SHUTDOWN command.

## States

Database is closed correctly

### State after using the command SHUTDOWN

- The `test.data` file is fully updated.
- The `test.backup` contains the compressed `test.data` file.
- The `test.script` contains the information in the database, excluding data for CACHED and TEXT tables.
- The `test.properties` contains the entry 'modified' set to 'no'.
- There is no `test.log` file.



Database is closed correctly with SHUTDOWN SCRIPT

### **State after using the command SHUTDOWN SCRIPT**

- The `test.data` file does not exist; all CACHED table data is in the `test.script` file
- The `test.backup` does not exist.
- The `test.script` contains the information in the database, including data for CACHED and TEXT tables.
- The `test.properties` contains the entry 'modified' set to 'no'.
- There is no `test.log` file.

Database is aborted

This may happen by sudden power off, Ctrl+C in Windows, but may be simulated using the command SHUTDOWN IMMEDIATELY.

### **Aborted Database state**

- The `test.properties` still contains 'modified=yes'.
- The `test.script` contains a snapshot of the database at the last checkpoint and is OK.
- The `test.data` file may be corrupt because the cache in memory was not written out completely.
- The `test.backup` file contains a snapshot of `test.data` that corresponds to `test.script`.
- The `test.log` file contain all information to re-do all changes since the snapshot. As a result of abnormal termination, this file may be partially corrupt.

## **Procedures**

The database engine performs the following procedures internally in different circumstances.

## **Clean Shutdown**

### **Procedure C.1. Clean Hsqldb database shutdown**

1. The `test.data` file is written completely (all the modified cached table rows are written out) and closed.
2. The `test.backup.new` is created (containing the compressed `test.data` file)
3. The file `test.script.new` is created using the information in the database (and thus shrinks because no UPDATE and DELETE statements; only INSERT).
4. The entry 'modified' in the properties file is set to 'yes-new-files'

5. The file `test.script` is deleted
6. The file `test.script.new` is renamed to `test.script`
7. The file `test.backup` is deleted
8. The file `test.backup.new` is renamed to `test.backup`
9. The entry 'modified' in the properties file is set to 'no'
10. The file `test.log` is deleted

## Startup

### Procedure C.2. Database is opened

1. Check if the database files are in use (by checking a special `test.lck` file).
2. See if the `test.properties` file exists, otherwise create it.
3. If the `test.properties` did not exist, then this is a new database. Create the empty `test.log` to append new commands.
4. If it is an existing database, check in the `test.properties` file if 'modified=yes'. This would mean last time it was not closed correctly, and thus the `test.data` file may be corrupted or incomplete. In this case the 'REPAIR' algorithm is executed (see below), before the database is opened normally.
5. Otherwise, if in the `test.properties` file 'modified=yes-new-files', then the (old) `test.backup` and `test.script` files are deleted and the new `test.script.new` file is renamed to `test.script`.
6. Open the `test.script` file and execute the commands.
7. Create the empty `test.log` to append new commands.

## Repair

The current `test.data` file is corrupt, but with the old `test.data` (from the `test.backup` file and `test.script`) and the current `test.log`, the database is made up-to-date. The database engine takes these steps:

### Procedure C.3. Database Repair

1. Restore the old `test.data` file from the backup (uncompress the `test.backup` and overwrite `test.data`).
2. Execute all commands in the `test.script` file.

3. Execute all commands in the `test.log` file. If due to corruption, an exception is thrown, the rest of the lines of command in the `test.log` file are ignored.
4. Close the database correctly (including a backup).

---

# Appendix D. Running Hsqldb with OpenOffice.org

Hermann Kienlein, EDV - Systeme Kienlein <hermann@kienlein.com>

Copyright 2003-2004 Hermann Kienlein. Permission is granted to distribute this document without any alteration under the terms of the HSQLDB license. Additional permission is granted to the HSQLDB Development Group to distribute this document with or without alterations under the terms of the HSQLDB license.

\$Date: 2004/04/22 09:25:28 \$

## Introduction

HSQLDB can now act as a Database with OpenOffice.org. This document is written to help you connecting and running HSQLDB out of OpenOffice.org in a simple way. Without user-management and only for your single-system.

If you have problems read the other available documents, because I will not write them here again. If you need a real DB-System with user-management and different rights for different users, read the other documents.

## Installing

I assume you have a running OpenOffice.org (OOo) and a JavaRuntimeEnvironment. So place the hsqldb-1.7.2.\*.zip file where you want on your disk and unpack it (I assume you have done this already).

## Setting up OpenOffice.org

Start OOo with a text document and go to the Database-Explorer (simply by pressing F4). In the left frame you see a tree-view with all known databases in OOo.

A right mouse-click opens a menu where you can manage your databases. So click on New Database and choose a name that you want to have inside OOo. I chose HSQLDB as name.

As connection-type choose JDBC and then switch to the JDBC-tab.

As Driver-Class insert `org.hsqldb.jdbcDriver` and as URL choose the following:

## On Windows

You can specify a directory where HSQLDB should store the info and data. Something like `jdbc:hsqldb:file:c:\javasrc\hsqldb-dev\databasename` (where `jdbc:` is written by OOo). The string `c:\javasrc\hsqldb-dev\databasename` works only on windows, but you can write this down as linux-path like `/javasrc/hsqldb-dev/databasename` too. Then HSQLDB takes the `c:` drive as root. This means this works only on `c:` for you.

The first is the directory-path and the `databasename` is the identifier for the database.

## On Linux

Choose a path as said for windows like `/opt/db/data`

As username take sa, this is the standard-administrator for HSQLDB.

Now click the OK-Button

Now OOo has to find your `hsqldb.jar` file. So go to options => security and insert the path to the `.jar` file. If you have problems, search the Online-help for JDBC. You then get help in your own language (this is generally quite better than my English, I think ;-)

If you cannot write to your Tables, OOo thinks that you don't have permission to write to HSQLDB. Then we tell OOo to ignore the `DriverPrivileges` because on our single-user-system we do not need them.

Because OOo is working on this, the next Step is only needed for systems without write - permission.

So we go to <http://dba.openoffice.org> and look at the `IgnoreDriverPrivileges.html` file in the `HowTo`-section. You find here a macro-code.

Open tools => macro in OOo to get the Basic-IDE. Here simple copy and paste the code and run the macro. You see a input-box where you only have to insert the name of your DB, in my example I have to insert HSQLDB, because I took this as name in OOo.

Note that if you change your OOo-DB name, you have to run this macro again!

Now we only have to stop and restart OOo. Be sure that you exit Quickstarter and all running processes too. On next OOo-Start you should have a running Database in OpenOffice.org.

---

# Appendix E. Hsqldb Test Utility

\$Date: 2004/06/18 13:04:05 \$

The `org.hsqldb.test` package contains a number of tests for various functions of the database engine. Among these, the `TestSelf` class performs the tests that are based on scripts. To run the tests, you should compile the `hsqldbtest.jar` target with Ant.

For `TestSelf`, a batch file is provided in the `testrun/hsqldb` directory, together with a set of `TestSelf*.txt` files. To start the application in Windows, type:

```
runtest TestSelf
```

The new version of `TestSelf` runs multiple SQL test files to test different SQL operations of the database. All files in the working directory with names matching `TestSelf*.txt` are processed in alphabetical order.

You can add your own scripts to test different series of SQL queries. The format of the `TestSelf*.txt` file is simple text, with some indentation and prefixes in the form of Java-style comments. The prefixes indicate what the expected result should be.

- Comment lines must start with `--` and are ignored
- Lines starting with spaces are the continuation of the previous line
- SQL statements with no prefix are simply executed.
- *The remaining items in this list exemplify use of the available command line-prefixes.*

•

```
/*c<rows>*/ SQL statement returning column count of <rows>
```

•

```
/*u<count>*/ SQL statement returning an update count equal to <count>
```

•

```
/*e*/ SQL statement that should produce an error when executing
```

•

```
/*r<string1>,<string2>*/ SQL statement returning a single row ResultSet equal to
```

•

```
/*r
  <string1>,<string2>
  <string1>,<string2>
  <string1>,<string2>
*/ SQL statement returning a multiple row ResultSet equal to the specified value
```

(note that the result set lines are indented).

See the TestSelf\*.txt files for actual examples.

---

# Appendix F. Database Manager

Fred Toussi, HSQLDB Development Group <ft@cluedup.com>  
\$Date: 2004/06/18 14:24:42 \$

## Brief Introduction

The Database Manager tool is a simple GUI database query tool with a tree display of the tables. Both AWT and SWING versions of the tool are available and work almost identically. The AWT version class name is `org.hsqldb.util.DatabaseManager`; the SWING version, `org.hsqldb.util.DatabaseManagerSwing`.

The AWT version of the database manager can be deployed as an applet in a browser. A demo HTML file with an embedded Database Manager is included in the `/demo` directory.

When the Database Manager is started, a dialogue allows you to enter the JDBC driver, URL, user and password for the new connection. A drop-down box, Type, offers preset values for JDBC driver and URL for most popular database engines, including HSQLDB. Once you have selected an item from this drop-down box, you should edit the URL to specify the details of the database or any additional properties to pass. You should also enter the username and password before clicking on the OK button.

The AWT connection dialogue allows you to save the settings for the connection you are about to make. You can then access the connection in future sessions. To save a connection setting, enter a name in the Setting Name box before clicking on the OK button. Next time the connection dialogue is displayed, the drop-down box labeled Recent will include the name for all the saved connection settings. When you select a name, the individual settings are displayed in the appropriate boxes.

The small Clr button next to the drop-down box allows you to clear all the saved settings. If you want to modify an existing setting, first select it from the drop-down box then modify any of the text boxes before making the connection. The modified values will be saved.



---

# Appendix G. Transfer Tool

Fred Toussi, HSQLDB Development Group <ft@cluedup.com>  
\$Date: 2004/06/18 14:24:42 \$

## Brief Introduction

Transfer Tool is a GUI program for transferring SQL schema and data from one JDBC source to another. Source and destination can be different database engines or different databases on the same server.

Transfer Tool works in two different modes. Direct transfer maintains a connection to both source and destination and performs the transfer. Dump and Restore mode is invoked once to transfer the data from the source to a text file (Dump), then again to transfer the data from the text file to the destination (Restore). With Dump and Restore, it is possible to make any changes to database object definitions and data prior to restoring it to the target.

Dump and Restore modes can be set via the command line with `-d (--dump)` or `-r (--restore)` options. Alternatively the Transfer Tool can be started with any of the three modes from the Database Manager's Tools menu.

The connection dialogue allows you to save the settings for the connection you are about to make. You can then access the connection in future sessions. These settings are shared with those from the Database Manager tool. See the appendix on Database Manager for details of the connection dialogue box.