HyperSQL Utilities Guide

Edited by , Blaine Simpson, and Fred Toussi

HyperSQL Utilities Guide

by , Blaine Simpson, and Fred Toussi

$Revision: 5063 $

Publication date 2015-06-29 22:28:08-0400

Table of Contents

List of Tables

List of Examples

Preface

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://hsqldb.org/web/hsqlSupport.html .

Available formats for this document

This document is available in several formats.

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

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

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

Table 1. Available formats of this document

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

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

Chapter 1. SqlTool

SqlTool Manual

Blaine Simpson, HSQL Development Group

$Revision: 5463 $
2015-06-29 22:28:08-0400

Try It

If you know how to type in a Java command at your shell command line, and you know at least the most basic SQL commands, then you know enough to benefit from SqlTool. You can play with Java system properties, PL variables, math, and other things by just executing this `sqltool-2.2.6.jar` file [http://search.maven.org/remotecontent?filepath=org/hsqldb/sqltool/2.2.6/sqltool-2.2.6.jar]. But SqlTool was made for JDBC, so you should download HyperSQL to have SqlTool automatically connect to a fully functional, pure Java database; or obtain a JDBC driver for any other SQL database that you have an account in.

If HyperSQL's `hsqldb.jar` resides in the same directory as the SqlTool jar file, then you can connect up to a HyperSQL instance from SqlTool just by specifying the JDBC URL, root account user name of SA and empty password, with the `\j` command. See the Switching Data Sources section below for details about `\j`.

Purpose, Coverage, Recent Changes in Behavior

Note

Due to many important improvements to SqlTool, both in terms of stability and features, all users of SqlTool are advised to use the latest version of SqlTool, even if your database instances run with an older HSQLDB version. How to do this is documented in the Accessing older HSQLDB Databases with SqlTool section below.

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 and automation. The following paragraphs explain in a general way why SqlTool is better than any existing tool for text-mode interactive SQL work, and for automated SQL tasks. Two important benefits which SqlTool shares with other pure Java JDBC tools is that users can use a consistent interface and syntax to interact with a huge variety of databases-- any database which supports JDBC; plus the tool itself runs on any Java platform. Instead of using `isql` for Sybase, `psql` for Postgresql, `Sql*plus` for Oracle, etc., you can use SqlTool for all of them. As far as I know, SqlTool is the only production-ready, pure Java, command-line, generic JDBC client. Several databases come with a command-line client with limited JDBC abilities (usually designed for use with just their specific database).

Use the In-Program Help!

The SqlTool commands and settings are intuitive once you are familiar with the usage idioms. This Guide does not attempt to list every SqlTool command and option available. When you want to know what SqlTool commands or options are available for a specific purpose, you need to list the commands of the appropriate type with the relevant "?" command. For example, as explained below, to see all Special commands, you would run `\?`; and to see all DSV export options, you run \x?.

SqlTool is purposefully not a Gui tool like Toad or DatabaseManager. There are many use cases where a Gui SQL tool would be better. Where automation is involved in any way, you really need a text client to at least test things

properly and usually to prototype and try things out. A command-line tool is really better for executing SQL scripts, any form of automation, direct-to-file fetching, and remote client usage. To clarify this last, if you have to do your SQL client work on a work server on the other side of a VPN connection, you will quickly appreciate the speed difference between text data transmission and graphical data transmission, even if using VNC or Remote Console. Another case would be where you are doing some repetitive or very structured work where variables or language features would be useful. Gui proponents may disagree with me, but scripting (of any sort) is more efficient than repetitive copy & pasting with a Gui editor. SqlTool starts up very quickly, and it takes up a tiny fraction of the RAM required to run a comparably complex Gui like Toad.

SqlTool is superior for interactive use because over many years it has evolved lots of features proven to be efficient for day-to-day use. Four concise in-program help commands (`\?`, `:?`, `*?` and `/?`) list all available commands of the corresponding type. SqlTool doesn't support up-arrow or other OOB escapes (due to basic Java I/O limitations), but it more than makes up for this limitation with macros, user variables, command-line history and recall, and command-line editing with extended Perl/Java regular expressions. The \d commands deliver JDBC metadata information as consistently as possible (in several cases, database-specific work-arounds are used to obtain the underlying data even though the database doesn't provide metadata according to the JDBC specs). Unlike server-side language features, the same feature set works for any database server. Database access details may be supplied on the command line, but day-to-day users will want to centralize JDBC connection details into a single, protected RC file. You can put connection details (username, password, URL, and other optional settings) for scores of target databases into your RC file, then connect to any of them whenever you want by just giving SqlTool the ID ("urlid") for that database. When you Execute SqlTool interactively, it behaves by default exactly as you would want it to. If errors occur, you are given specific error messages and you can decide whether to roll back your session. You can easily change this behavior to auto-commit, exit-upon-error, etc., for the current session or for all interactive invocations. You can import or export delimiter-separated-value files. If you need to run a specific statement repeatedly, perhaps changing the WHERE clause each time, it is very simple to define a macro.

When you Execute SqlTool with a SQL script, it also behaves by default exactly as you would want it to. If any error is encountered, the connection will be rolled back, then SqlTool will exit with an error exit value. If you wish, you can detect and handle error (or other) conditions yourself. For scripts expected to produce errors (like many scripts provided by database vendors), you can have SqlTool continue-upon-error. For SQL script-writers, you will have access to portable scripting features which you've had to live without until now. You can use variables set on the command line or in your script. You can handle specific errors based on the output of SQL commands or of your variables. You can chain SQL scripts, invoke external programs, dump data to files, use prepared statements, Finally, you have a procedural language with `if`, `foreach`, `while`, `continue`, and `break` statements.

Platforms and SqlTool versions covered

SqlTool runs on any Java 1.5 or later platform. I know that SqlTool works well with Sun and OpenJDK JVMs. I haven't run other vendors' JVMs in years (IBM, JRockit, etc.). As my use with OpenJDK proves that I don't depend on Sun-specific classes, I expect it to work well with other (1.5-compatible) Java implementations.

SqlTool no longer writes any files without being explicitly instructed to. Therefore, it should work fine on read-only systems, and you'll never have orphaned temp files left around.

The command-line examples in this chapter work as given on all platforms (if you substitute in a normalized path in place of `$HSQLDB_HOME`), except where noted otherwise. When doing any significant command-line work on Windows (especially shell scripting), you're better off to completely avoid paths with spaces or funny characters. If you can't avoid it, use double-quotes and expect problems. As with any Java program, file or directory paths on the command line after "java" can use forward slashes instead of back slashes (this goes for System properties and the `CLASSPATH` variable too). I use forward slashes because they can be used consistently, and I don't have to contort my fingers to type them :).

If you are using SqlTool from a HyperSQL distribution of version 2.2.5 or earlier, you should use the documentation with that distribution, because this manual documents many new features, several significant changes to interactive-

only commands, and a few changes effecting backwards-compatibility (see next section about that). This document is now updated for the current versions of SqlTool and SqlFile at the time I am writing this (versions 5337 and 5448 correspondingly-- SqlFile is the class which actually processes the SQL content for SqlTool). Therefore, if you are using a version of SqlTool or SqlFile that is more than a couple revisions greater, you should find a newer version of this document. (The imprecision is due to content-independent revision increments at build time, and the likelihood of one or two behavior-independent bug fixes after public releases). The startup banner will report both versions when you run SqlTool interactively. (Dotted version numbers of SqlTool and SqlFile definitely indicate ancient versions).

This guide covers SqlTool as bundled with HSQLDB after 2.2.5. [1]

Recent Functional Changes

This section lists changes to SqlTool since the last major release of HSQLDB which may effect the portability of SQL scripts. For this revision of this document, this list consists of script-impacting changes made to SqlTool *after* the final 2.0.0 HyperSQL release.

- SqlTool always has treated unset PL variables equal to null-valued variables, and this is not changing. There is no distinction between *unset* and *null-valued*. But before revision 4423 (i.e. pre-HyperSQL 2.2.6), SqlTool almost always treated variable value of empty String (i.e. what is between these two quotes: "") as equivalent to null/unset. The most common case where a user would see this was in the command to *unset* a PL variable: `* VARNAME =`. Though your command says to assign the value of the empty string to `VARNAME`, varname would unset it instead (which latter is equivalent to assigning null to it).

  SqlTool achieves several new benefits by purposefully assigning the empty string for some purposes, and distinctly unsetting for other purposes. One example of this is the consistent and robust handling of the PL variable `?`. When `?` is unset (aka *null*), it always means that the last SQL command failed. If this variable has the value of the empty string, it means that the last cell returned had value of the empty string, or the `?` was reset (for example due to SqlTool startup).

  For now on (in accordance with years of warnings) by default, setting a PL variable to nothing, like `* name =` assigns an empty-string value instead of unsetting/removing the variable. To unset/remove a variable you should use the dedicated PL command "`* - VARNAME`". You can set Java system property '`sqltool.REMOVE_EMPTY_VARS`' to keep the inferior, legacy behavior where `* name =` will unset the variable (even with this setting you can still intermix this with the `* - VARNAME` command) . You should only use '`sqltool.REMOVE_EMPTY_VARS`' in conjunction with old scripts that you can't update.

  You can use command line parameters `-P` or `--setVar` to assign PL variables to the empty String. (For example "`java -jar .../sqltool.jar -Pvarname= urlid script.sql`").

- DSV input now accepts JDBC Timestamp format with date and optionally time of day.

- The `*` command (just plain "`*`") has been removed because it is no longer necessary. (See item on it in the previous section of this manual).

- Bugs in variable scoping have been fixed. All variables are global and shared among `auto.sql`, all command-line and nested SQL scripts, and interactive shells.

- A bug has been fixed so that SqlTool system PL variables (with names starting with "*") are now honored no matter how or where the variable value was set.

- Variables with non-alpha-numerical characters, and those beginning with digits have been deprecated but are still supported for now.

[1] To reduce the time I will need to spend maintaining this document, in this chapter I am giving the path to the `sample` directory as it is in HyperSQL 2.0.x distributions, namely, `HSQLDB_HOME/sample`. Users of HSQLDB before 2.0.x should translate these sample directory paths to use `HSQLDB_HOME/src/org/hsqldb/sample/...`.

- The data (and just the data) of HTML reports has remained the same, but the whole system has been drastically modernized and enhanced. You may notice new special command `\pr`. This is a `\p` variant applicable only to HTML mode, telling SqlTool to treat the expression as Raw, so the author can type in HTML, JavaScript or other text not to be formatted. HTML model also gets its own `*NULL_REP_HTML` setting distinct from `*NULL_REP_TOKEN`.

- For consistency, the \H toggle switch has been superseded with `\h true|false`> The old variants are still supported.

- Command `/= name :...` has been superseded by the more consistent (with our other commands) `/: name...`. The old variant is still supported.

- Removed support for SqlTool system PL variable names deprecated years ago.

- Idiosyncratic `*` option to `\m` is no longer necessary (though still supported). SqlTool system variable `*DSV_SKIP_PREFIX` may now be set to the empty string to indicate no skipping.

- Import reject report files were being retained on Windows, even when there were not rejected records. This bug has been fixed.

- PL variables `*NULL` and `NULL` are reserved and may be referenced but not assigned to.

- The special PL variable `?` is now updated rigorously. It will be null only if the last SQL command that was run failed.

- The special PL variable `?` is now listed in the output of `* list` and `* listvalues` commands. (Unless at the time it is unset/null, of course, and in this case the absence of `?` indicates that it is unset.

- The look of DSV import reject reports has changed, and the name of user-supplied (optional) CSS file has been changed. (See following section about why).

- The `\d tableLikeObject` command output now has 2 more columns, to report precision and scale.

New Features

Since 2.0.0 final

To reduce duplication, new features are listed in the Recent Functional Changes section are not repeated here, so check that list too.

- Automatic variable # added.

- You can (and if you use nested scripts, you probably *should*) prefix relative paths given inside of SqlTool with `@/`. This makes them relative to the parent script directory instead of to the invocation current directory.

- More accepting of dates and times in DSV imports.

- CSV exporting and importing with double-quote delimiting and escaping. Fixed a bug that caused SqlTool to abort if it attempted to load a doube-quoted CSV field containing just one character of data.

- New "`* - VARNAME`" PL command added to explicitly unset/remove a variable. Definitely read the first item in the Recent Functional Changes section.

- Added integer math feature very similar to that of Bash and Korn shells.

- Added some new logical operators

- Added new : variant for exporting (\x and \xq), which uses a query from the edit buffer. This provides direct support for exports using long, multi-line queries.
- SqlTool functions have been implemented. These are just / macros which take positional parameters. They can be distinguished from regular macros by having name like `this()`, and being invoked like `this(with, parameters)`.
- We have provided multiple ways to accommodate command-line *inlineRc*s and variable assignments where the values contain commas, and generally added flexibility to the latter. In both cases, commas may be escaped with the backslash character. New switch -p (also usable as -P) is provided as an easier way to eliminate the delimiter issue.
- Multiple --sql, -p, and -P arguments to SqlTool are now honored, and they are evaluated in specified order.
- Added automatically-assigned SqlTool system PL variables `*START_TIME` and `*REVISION`.
- Added optional SqlTool system PL variable `*TIMESTAMP_FORMAT`, which, when set causes new SqlTool system PL variable `*TIMESTAMP` to be automatically set to the time when this (latter) variable is referenced. Unlike `*START_TIME`, the `*TIMESTAMP_FORMAT` + `*TIMESTAMP` combination allows the user to specify exactly how to display the time and/or date.
- The DSV import reject report has been refactored to use the same templating and substitution used by the new HTML reports.
- The SqlTool unit test system has been ported from Bash to Groovy wrapped by Gradle. The system requirements have thereby been reduced from requiring a Bash shell to just requiring a Java 6 JRE.
- Added an inline/shortcut `*if` syntax like this: `* if (*x == *b) \p Any 1-lne statement`. People who value conciseness, like myself, will appreciate this.
- Added `* else` statement to accompany `* if` statement.
- Added traditional mathematical `* if` statement. Run `*?` to see syntax.
- Added iteration over query result set rows with loop statement `* forrows`.
- Added statement `* return` as a more intuitive alias for `* break file`.

The Bare Minimum

The Bare Minimum You Need to Know to Run SqlTool

Warning

If you are using an Oracle database server, it will commit your current transaction if you cleanly disconnect, regardless of whether you have set auto-commit or not. This will occur if you exit SqlTool (or any other client) in the normal way (as opposed to killing the process or using Ctrl-C, etc.). This is mentioned in this section only for brevity, so I don't need to mention it in the main text in the many places where auto-commit is discussed. This behavior has nothing to do with SqlTool. It is a quirk of Oracle.

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 probably looking at the wrong program.

Procedure 1.1. To run SqlTool...

1. Copy the file `sample/sqltool.rc` [1] of your HyperSQL distribution to your home directory and secure access to it if your computer is accessible to anybody else (most likely from the network). This file will work as-is

for a Memory Only database instance; or if your target is a HyperSQL 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 HyperSQL 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 RC File Authentication Setup section. You could, alternatively, use the `--inlineRc` command-line switch or the \j special command to connect up to a data source, as documented below.

2. Find out where your `sqltool.jar` file resides. It typically resides at `HSQLDB_HOME /lib/sqltool.jar` where `HSQLDB_HOME` is the "hsqldb" directory inside the root level of your HyperSQL software installation. (For example, if you extract `hsqldb-9.1.0.zip` into `c:\temp`, your `HSQLDB_HOME` would be `c:/temp/hsqldb-9.1.0/hsqldb`. Your file may also have a version label in the file name, like `sqltool-1.2.3.4.jar`. The forward slashes work just fine on Windows). For this reason, I'm going to use "$HSQLDB_HOME/lib/sqltool.jar" as the path to `sqltool.jar` for my examples, but understand that you need to use the actual path to your own `sqltool.jar` file. (Unix users may set a real env. variable if they wish, in which case the examples may be used verbatim; Window users may do the same, but will need to dereference the variables like `%THIS%` instead of like `$THIS`).

Warning

My examples assume there are no spaces or funky characters in your file paths. This avoids bugs with the Windows cmd shell and makes for simpler syntax all-around. If you insist on using directories with spaces or shell metacharacters (including standard Windows home directories like `C:\Documents and Settings\blaine`), you will need to double-quote arguments containing these paths. (On UNIX you can alternatively use single-quotes to avoid variable dereferencing at the same time).

3. If you are just starting with SqlTool, you are best off running your SqlTool command from a shell *command-line* (as opposed to by using icons or the Windows' Start/Run... or Start/Start Search). This way, you will be sure to see error messages if you type the command wrong or if SqlTool can't start up for some reason. On recent versions of Windows, you can get a shell by running `cmd` from Start/Run... or Start/Start Search). On UNIX or Linux, any real or virtual terminal will work.

   On your shell command line, run

```
java -jar $HSQLDB_HOME/lib/sqltool.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`.

   To run SqlTool without a JDBC connection, run

```
java -jar $HSQLDB_HOME/lib/sqltool.jar
```

   You won't be able to execute any SQL, but you can play with the SqlTool interface (including using PL features).

   To execute SQL, you'll need the classes for the target database's JDBC driver (and database engine classes for *in-process* databases). As this section is titled *The Bare Minimum*, I'll just say that if you are running SqlTool from a HyperSQL product installation, you are all set to connect to any kind of HyperSQL database. This is because SqlTool will look for the file `hsqldb.jar` in the same directory as `sqltool.jar`, and that file contains all of the needed classes. (SqlTool supports all JDBC databases and does not require a HyperSQL installation, but these cases would take us beyond *the bare minimum*). So, with `hsqldb.jar` in place, you can run commands like

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

   for interactive use, or

```
java -jar $HSQLDB_HOME/lib/sqltool.jar --sql="SQL statement;" mem
```

or

```
java -jar $HSQLDB_HOME/lib/sqltool.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 RC file, as explained in the RC File Authentication Setup section. Since this is a *mem:* type catalog, 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 HyperSQL Listener. At the end of this section, I explain how you can load some sample data to play with, if you want to.

Tip

If SqlTool fails to connect to the specified urlid and you don't know why, add the invocation parameter `--debug`. This will cause SqlTool to display a stack trace from where the connection attempt fails. (If a connection attempt fails with the interactive \j command, details will always be displayed).

You are responsible for Commit behavior

SqlTool does not *commit* SQL changes by default. (You can use the `--autoCommit` command-line switch to have it auto-commit). This leaves it to the user's discretion whether to commit or rollback their modifications. If you do want your changes committed, remember to run \= before quitting SqlTool. (Most databases also support the SQL command `commit;`),

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 (unless you invoke with the `--noAutoFile` switch). I did say *interactively*: If you want to execute this file when you execute SQL scripts from the command line, then your script must use `\i ${user.home}/auto.sql` or similar to invoke it explicitly.

To use a JDBC Driver other than the HyperSQL driver, you can't use the `-jar` switch because you need to modify the classpath. You must add the `sqltool.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 RC File 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.cmdline.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).

Quotes and Spaces

Single and double-quotes are not treated specially by SqlTool. This makes SqlTool more intuitive than most shell languages, ensures that quotes sent to the database engine are not adulterated, and eliminates the need for somehow *escaping* quote characters.

Line delimiters are special, as that is the primary means for SqlTool to tell when a command is finished (requiring combination with semi-colon to support multi-line SQL statements). Spaces and tabs are preserved inside of your strings and variable values, but are trimmed from the beginning in nearly all cases (such space having very rare usefulness). The cases where leading whitespace is preserved exactly as specified in your strings are the : commands (including `* VARNAME :`, `/: VARNAME`, `\x :`, and `\xq :`).

So, if you write the SQL command

```
INSERT into t values ('one ''    and   ''  two');
```

or the SqlTool print command

```
\p A message for my 'Greatest...     fan'
```

you just type exactly what you want to send to the database, or what you want displayed.

Embedding

Using SqlTool to execute SQL files from your own Java code

To repeat what is stated in the JavaDoc for the `SqlTool` class itself: *Programmatic users will usually want to use the objectMain(String[]) method if they want arguments and behavior exactly like command-line SqlTool. If you don't need invocation parameter parsing, `auto.sql` execution, etc., you will have more control and efficiency by using the SqlFile class directly. The file `src/org/hsqldb/sample/SqlFileEmbedder.java` in the HyperSQL distribution provides an example for this latter strategy.*

Non-displayable Types

There are some SQL types which SqlTool (being a text-based program) can't display properly. This includes the SQL types `BLOB`, `JAVA_OBJECT`, `STRUCT`, and `OTHER`. When you run a query that returns any of these, SqlTool will save the very first such value obtained to the binary buffer and will not display any output from this query. You can then save the binary value to a file, as explained in the Storing and Retrieving Binary Files section.

There are other types, such as `BINARY`, which JDBC can make displayable (by using ResultSet.getString()), but which you may very well want to retrieve in raw binary format. You can use the \b command to retrieve any-column-type-at-all in raw binary format (so you can later store the value to a binary file).

Another restriction which all text-based database clients have is the practical inability for the user to type in binary data such as photos, audio streams, and serialized Java objects. You can use SqlTool to load any binary object into a database by telling SqlTool to get the insert/update datum from a file. This is also explained in the Storing and Retrieving Binary Files section.

Compound commands or commands with semi-colons

See the Chunking section if you need to execute any compound SQL commands or SQL commands containing non-escaped/quoted semi-colons.

Desktop shortcuts

Desktop shortcuts and quick launch icons are useful, especially if you often run SqlTool with the same set of arguments. It's really easy to set up several of them-- one for each way that you invoke SqlTool (i.e., each one would start SqlTool with all the arguments for one of your typical startup needs). One typical setup is to have one shortcut for each database account which you normally use (use a different `urlid` argument in each shortcut's Target specification.

Desktop icon setup varies depending on your Desktop manager, of course. I'll explain how to set up a SqlTool startup icon in Windows XP. Linux and Mac users should be able to take it from there, since it's easier with the common Linux and Mac desktops.

Procedure 1.2. Creating a Desktop Shortcut for SqlTool

1. Right click in the main Windows background.
2. New
3. Shortcut
4. Browse
5. Navigate to where your good JRE lives. For recent Sun JRE's, it installs to `C:\Program Files\Java\*\bin` by default (the * will be a JDK or JRE identifier and version number).
6. Select `java.exe`.
7. OK
8. Next
9. Enter any name
10. Finish
11. Right click the new icon.
12. Properties
13. Edit the Target field.
14. Leave the path to java.exe exactly as it is, including the quotes, but append to what is there. Beginning with a space, enter the command-line that you want run.
15. Change Icon... to a pretty icon.
16. If you want a quick-launch icon instead of (or in addition to) a desktop shortcut icon, click and drag it to your quick launch bar. (You may or may not need to edit the Windows Toolbar properties to let you add new items). Postnote: Quick launch setup has become more idiosyncratic on the more recent versions of Windows, sometimes requiring esoteric hacks to make them in some cases. So, if the instructions here don't work, you'll have to seek help elsewhere.

Loading sample data

If you want some sample database objects and data to play with, execute the `sample/sampledata.sql` SQL file [1]. To separate the sample data from your regular data, you can put it into its own schema by running this before you import:

```
   CREATE SCHEMA sampledata AUTHORIZATION dba;
SET SCHEMA sampledata;
```

Run it like this from an SqlTool session

```
   \i HSQLDB_HOME/sample/sampledata.sql
```

where **HSQLDB_HOME** is the base directory of your HSQLDB software installation [1].

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.

Satisfying SqlTool's CLASSPATH Requirements

As discussed earlier, only the single file `sqltool.jar` is required to run SqlTool (the file name may contain a version label like `sqltool-1.2.3.4.jar`). But it's useless as an SQL *Tool* unless you can connect to a JDBC data source, and for that you need the target database's JDBC driver in the classpath. For *in-process* catalogs, you'll also need the database engine classes in the CLASSPATH. The The Bare Minimum section explains that the easiest way to use SqlTool with any HyperSQL database is to just use `sqltool.jar` in-place where it resides in a HyperSQL installation. This section explains how to satisfy the CLASSPATH requirements for other setups and use cases.

Accessing older HSQLDB Databases with SqlTool

If you are using SqlTool to access non-HSQLDB database(s), then you should use the latest and greatest-- just grab the newest public release of SqlTool (like from the latest public HyperSQL release) and skip this subsection.

You are strongly encouraged to use the latest SqlTool release to access older HSQLDB databases, to enjoy greatly improved SqlTool robustness and features. It is very easy to do this.

1. Obtain the latest `sqltool.jar` file. One way to obtain the latest `sqltool.jar` file is to download the latest HyperSQL distribution and extract that single file

2. Place (or copy) your new `sqltool.jar` file right alongside the `hsqldb.jar` file for your target database version. If you don't have a local copy of the `hsqldb.jar` file for your target database, just copy it from your database server, or download the full distribution for that server version and extract it.

3. *(If you have used older versions of SqlTool before, notice that you now invoke SqlTool by specifying the `sqltool.jar` file instead of the `hsqldb.jar`).* If your target database is a previous 2.x version of HyperSQL, then you are finished and can use the new SqlTool for your older database. Users upgrading from a pre-2.x version please continue...

   Run SqlTool like this.

   ```
   java -jar path/to/sqltool.jar --driver=org.hsqldb.jdbcDriver...
   ```

   where you specify the pre-2.x JDBC driver name `org.hsqldb.jdbcDriver`. Give any other SqlTool parameters as you usually would.

   Once you have verified that you can access your database using the `--driver` parameter as explained above, edit your `sqltool.rc` file, and add a new line

   ```
   driver org.hsqldb.jdbcDriver
   ```

   after each urlid that is for a pre-2.x database. Once you do this, you can invoke SqlTool as usual (i.e. you no longer need the `--driver` argument for your invocations).

App-specific Classes, Embedding, and non-HyperSQL Databases

For these situations, you need to add your custom, third-party, or SQL driver classes to your Java CLASSPATH. Java doesn't support adding arbitrary elements to the classpath when you use the `-jar`, so you must set a classpath

containing `sqltool.jar` plus whatever else you need, then invoke SqlTool without the `-jar` switch, as briefly described at the end of the The Bare Minimum section. For embedded apps, invoke your own main class instead of SqlTool, and you can invoke `SqlTool` or `SqlFile` from your code base.

To customize the classpath, you need to set up your classpath by using your operating system or shell variable `CLASSPATH` or by using the `java` switch `-cp` (or the equivalent `-classpath`). I'm not going to take up space here to explain how to set up a Java CLASSPATH. That is a platform-dependent task that is documented well in tons of Java introductions and tutorials. What I'm responsible for telling you is *what* you need to add to your classpath. For the non-embedded case where you have set up your CLASSPATH environmental variable, you would invoke SqlTool like this.

```
    java org.hsqldb.cmdline.SqlTool ...
```

If you are using the `-cp` switch instead of a `CLASSPATH` variable, stick it after `java`. After "`SqlTool`", give any SqlTool parameters exactly as you would put after `java -jar .../sqltool.jar` if you didn't need to customize the CLASSPATH. You can specify a JDBC driver class to use either with the `--driver` switch to SqlTool, or in your RC file stanza (the last method is usually more convenient).

Note that without the `-jar` switch, SqlTool will still automatically pull in HyperSQL JDBC driver or engine classes from HyperSQL jar files in the same directory. It's often a good practice to minimize your runtime classpath. To prevent the possibility of pulling in classes from other HyperSQL jar files, just copy `sqltool.jar` to some other directory (which does not contain other HyperSQL jar files) and put the path to that one in your classpath.

Distributing SqlTool with your Apps

You can distribute SqlTool along with your application, for standalone or embedded invocation. For embedded use, you will need to customize the classpath as discussed in the previous item. Either way, you should minimize your application footprint by distributing only those HyperSQL jar files needed by your app. You will obviously need `sqltool.jar` if you will use the `SqlTool` or `SqlFile` class in any way. If your app will only connect to external HyperSQL listeners, then build and include `hsqljdbc.jar`. If your app will also *run* a HyperSQL Listener, you'll need to include `hsqldb.jar`. If your app will connect directly to a *in-process* catalog, then include `hsqldbmain.jar`. Note that you never need to include more than one of `hsqldb.jar`, `hsqldbmain.jar`, `hsqljdbc.jar`, since the former jars include everything in the following jars.

SqlTool Client PCs

If you just want to be able to run SqlTool (interactively or non-interactively) on a PC, and have no need for documentation, then it's usually easiest to just copy `sqltool.jar` and `hsqldb.jar` to the PCs (plus JDBC driver jars for any other target databases). If you want to minimize what you distribute, then build and distribute `hsqljdbc.jar` or `hsqldbmain.jar` instead of `hsqldb.jar`, according to the criteria listed in the previous sub-section.

RC File Authentication Setup

RC file authentication setup is accomplished by creating a text RC configuration file. In this section, when I say *configuration* or *config* file, I mean an RC configuration file. RC files can be used by any JDBC client program that uses the org.hsqldb.util.RCData class-- this includes SqlTool, DatabaseManager, DatabaseManagerSwing.

You can use it for your own JDBC client programs too. There is example code showing how to do this at `src/org/hsqldb/sample/SqlFileEmbedder.java` .

The sample RC file shown here resides at `sample/sqltool.rc` in your HSQLDB distribution [1].

Example 1.1. Sample RC File

```
# $Id: sqltool.rc 5288 2013-09-29 18:35:42Z unsaved $

# This is a sample RC configuration file used by SqlTool, DatabaseManager,
# and any other program that uses the org.hsqldb.lib.RCData class.
# See the documentation for SqlTool for various ways to use this file.

# If you have the least concerns about security, then secure access to
# your RC file.

# You can run SqlTool right now by copying this file to your home directory
# and running
#    java -jar /path/to/sqltool.jar mem
# This will access the first urlid definition below in order to use a
# personal Memory-Only database.
# "url" values may, of course, contain JDBC connection properties, delimited
# with semicolons.
# As of revision 3347 of SqlFile, you can also connect to datasources defined
# here from within an SqlTool session/file with the command "\j urlid".

# You can use Java system property values in this file like this:  ${user.home}

# The only feature added recently is the optional "transiso" setting,
# which may be set to an all-caps transaction isolation level as listed
# in the Java API Spec for java.sql.Connection.
# Windows users are advised to use forward slashes instead of reverse slashes,
# and to avoid paths containing spaces or other funny characters.  (This
# recommendation applies to any Java app, not just SqlTool).

# A personal Memory-Only (non-persistent) database.
urlid mem
url jdbc:hsqldb:mem:memdbid
username SA
password

# A personal, local, persistent database.
urlid personal
url jdbc:hsqldb:file:${user.home}/db/personal;shutdown=true
username SA
password
transiso TRANSACTION_READ_COMMITTED
# When connecting directly to a file database like this, you should
# use the shutdown connection property like this to shut down the DB
# properly when you exit the JVM.

# This is for a hsqldb Server running with default settings on your local
# computer (and for which you have not changed the password for "SA").
urlid localhost-sa
url jdbc:hsqldb:hsql://localhost
username SA
password



# Template for a urlid for an Oracle database.
# You will need to put the oracle.jdbc.OracleDriver class into your
# classpath.
# In the great majority of cases, you want to use jhe desired version of a
# file odbc*.jar (previously JDBC distributed as 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 .../sqltool.jar...".
# Put both the SqlTool jar and odbc*.jar in your classpath (and export!)
# and run something like "java org.hsqldb.util.SqlTool...".
# You could use the thick driver instead of the thin, but I know of no reason
```

```
# why any Java app should.

#urlid cardiff2
#url jdbc:oracle:thin:@aegir.admc.com:1521:TRAFFIC_SID
# Thin SID URLs must specify both port and SID, there are no defaults.
# Oracle listens to 1521 by default, so that's what you will usually specify.
# But can alternatively use global service name (not tnsnames.ora service
# alias, in which case the port does default to 1521):
#url jdbc:oracle:thin:@centos.admc.com/tstsid.admc
#username blaine
#password secretpassword
#driver oracle.jdbc.OracleDriver



# Template for a TLS-encrypted HSQLDB Server.
# Remember that the hostname in hsqls (and https) JDBC URLs must match the
# CN of the server certificate (the port and instance alias that follows
# are not part of the certificate at all).
# You only need to set "truststore" if the server cert is not approved by
# your system default truststore (which a commercial certificate probably
# would be).

#urlid tls
#url jdbc:hsqldb:hsqls://db.admc.com:9001/lm2
#username BLAINE
#password asecret
#truststore ${user.home}/ca/db/db-trust.store


# Template for a Postgresql database
#urlid blainedb
#url jdbc:postgresql://idun.africawork.org/blainedb
#username blaine
#password losung1
#driver org.postgresql.Driver

# Template for a MySQL database.  MySQL has poor JDBC support.
#urlid mysql-testdb
#url jdbc:mysql://hostname:3306/dbname
#username root
#password hiddenpwd
#driver com.mysql.jdbc.Driver

# Note that "databases" in SQL Server and Sybase are traditionally used for
# the same purpose as "schemas" with more SQL-compliant databases.

# Template for a Microsoft SQL Server database using Microsoft's Driver
# (I find that the JTDS driver is much more responsive than Microsoft's).
# Port defaults to 1433.
# MSDN implies instances are port-specific, so can specify port or instname.
#urlid msprojsvr
# url/driver for Current 2011 JDBC Driver for Microsoft SQL Server:
# Requires just the new sqljdbc4.jar.  (Microsoft just loves back-slashes)
#url jdbc:sqlserver://hostname\instname;databaseName=dbname
# OR
#url jdbc:sqlserver://hostname;instanceName=instname;databaseName=dbname
#driver com.microsoft.jdbc.sqlserver.SQLServerDriver
# url/deriver for OLDER JDBC Driver:
#url jdbc:microsoft:sqlserver://hostname;DatabaseName=DbName;SelectMethod=Cursor
# The SelectMethod setting is required to do more than one thing on a JDBC
# session (I guess Microsoft thought nobody would really use Java for
# anything other than a "hello world" program).
# This is for Microsoft's SQL Server 2000 driver (requires mssqlserver.jar
# and msutil.jar).
#driver com.microsoft.jdbc.sqlserver.SQLServerDriver
#username myuser
```

```
#password hiddenpwd

# Template for Microsoft SQL Server database using the JTDS Driver
# http://jtds.sourceforge.net  Jar file has name like "jtds-1.2.5.jar".
# Port defaults to 1433.
# MSDN implies instances are port-specific, so can specify port or instname.
#urlid nlyte
#username myuser
#password hiddenpwd
#url jdbc:jtds:sqlserver://myhost/nlyte;instance=MSSQLSERVER
# Where database is 'nlyte' and instance is 'MSSQLSERVER'.
# N.b. this is diff. from MS tools and JDBC driver where (depending on which
# document you read), instance or database X are specified like  HOSTNAME\X.
#driver net.sourceforge.jtds.jdbc.Driver

# Template for a Sybase database
#urlid sybase
#url jdbc:sybase:Tds:hostname:4100/dbname
#username blaine
#password hiddenpwd
# This is for the jConnect driver (requires jconn3.jar).
#driver com.sybase.jdbc3.jdbc.SybDriver

# Template for Embedded Derby / Java DB.
#urlid derby1
#url jdbc:derby:path/to/derby/directory;create=true
#username ${user.name}
#password any_noauthbydefault
#driver org.apache.derby.jdbc.EmbeddedDriver
# The embedded Derby driver requires derby.jar.
# There'a also the org.apache.derby.jdbc.ClientDriver driver with URL
# like jdbc:derby://<server>[:<port>]/databaseName, which requires
# derbyclient.jar.
# You can use \= to commit, since the Derby team decided (why???)
# not to implement the SQL standard statement "commit"!!
# Note that SqlTool can not shut down an embedded Derby database properly,
# since that requires an additional SQL connection just for that purpose.
# However, I've never lost data by not shutting it down properly.
# Other than not supporting this quirk of Derby, SqlTool is miles ahead of ij.
```

As noted in the comment (and as used in a couple examples), you can use Java system properties like this: `${user.home}`. Windows users, please read the suggestion directed to you in the file.

You can put this file anywhere you want to, and specify the location to SqlTool/DatabaseManager/DatabaseManagerSwing 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/DatabaseManager/DatabaseManagerSwing. The default location is `sqltool.rc` or `dbmanager.rc` in your home directory (corresponding to the program using it). If you have any doubt about where your home directory is, 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/sqltool.jar x
```

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

```
    urlid web
    url jdbc:hsqldb:hsql://localhost
    username web
    password webspassword
```

These four settings are required for every urlid. (There are optional settings also, which are described a couple paragraphs down). The URL may contain JDBC connection properties. 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/DatabaseManager command must match a *urlid* in your configuration file.

Warning

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

It should be readable, both locally and remotely, only to users who run programs that need it. 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 — This is used by the SqlTool program, but not by the DatabaseManager programs. See the Character Encoding section of the Non-Interactive section. This is used for input and output files, not for stdin or stdout, which are controlled by environmental variables and Java system properties. If you set no encoding for an urlid, input and outfiles will use the same encoding as for stdin/stdout. (As of right now, the charset setting here is not honored by the \j command, but only when SqlTool loads an urlid specified on the command-line).

driver — Sets the JDBC driver class name. You can, alternatively, set this for one SqlTool/DatabaseManager invocation by using the command line switch *--driver*. Defaults to *org.hsqldb.jdbc.JDBCDriver*.

truststore — TLS trust keystore store file path as documented in the TLS section of the Listeners chapter of the HyperSQL User Guide [http://hsqldb.org/doc/2.0/guide/index.html] 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). Relative paths will be resolved relative to the `${user.dir}` system property at JRE invocation time.

transiso — Specify the Transaction Isolation Level with an all-caps string, exactly as listed in he Field Summary of the Java API Spec for the class `java.sql.Connection`.

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

Switching Data Sources

The \j command lets you switch JDBC Data Sources in your SQL files (or interactively). "\?" shows the syntax to make a connection by either RCData urlid or by name + password + JDBC Url. (If you omit the password parameter, an empty string password will be used). The urlid variant uses RC file of `$HOME/sqltool.rc`. We will add a way to specify an RC file if there is any demand for that.

You can start SqlTool without any JDBC Connection by specifying no Inline RC and urlid of "-" (just a hyphen). If you don't need to specify any SQL file paths, you can skip the hyphen, as in this example.

```
java -jar $HSQLDB_HOME/lib/sqltool.jar -Pv1=one
```

(The "-" is required when specifying one or more SQL files, in order to distinguish urlid-spec from file-spec). Consequently, if you invoke SqlTool with no parameters at all, you will get a SqlTool session with no JDBC Connection. You will obviously need to use \j before doing any database work.

Using Inline RC Authentication

Inline RC authentication setup is accomplished by using the `--inlineRc` command-line switch on SqlTool. The `--inlineRc` command-line switch takes a comma-separated list of key/value elements. The `url` and `user` elements

are required. The rest are optional. The `--inlineRc` switch is the only case where you can give SQL file paths without a preceding urlid indicator (an urlid or -). The program knows not to look for an urlid if you give an inline.

Since commas are used to separate each `name=value` pair, you must do some extra work for any commas inside of the *values* of any `name=value`s. Escape them by proceeding them with backslash, like "`myName=my\p,value`" to inform SqlTool that the comma is part of the value and not a name/value separator.

`url` The JDBC URL of the database you wish to connect to.

`user` The username to connect to the database as.

`charset` Sets the character encoding. Overrides the platform default, or what you have set by env variables or Java system properties. (Does not effect stdin or stdout).

`truststore` The TLS trust keystore file path as documented in the TLS chapter. Relative paths will be resolved relative to the current directory.

`transiso` `java.sql.Connection` transaction isolation level to connect with, as specified in the Java API spec.

`password` You may only use this element to set empty password, like

```
    password=
```

For any other password value, omit the `password` element and you will be prompted for the value.

(Use the `--driver` switch instead of `--inlineRc` to specify a JDBC driver class). Here is an example of invoking SqlTool to connect to a standalone database.

```
java -jar $HSQLDB_HOME/lib/sqltool.jar --inlineRc=url=jdbc:hsqldb:file:/home/dan/dandb,user=dan
```

For security reasons, you cannot specify a non-empty password as an argument. You will be prompted for a password as part of the login process.

Logging

Both the \l command and all warnings and error messages now use a logging facility. The logging facility hands off to Log4j if Log4j is found in the classpath, and otherwise will hand off to `java.util.logging`. The default behavior of `java.util.logging` should work fine for most users. If you are using log4j and are redirecting with pipes, you may want to configure a Console Appender with target of `"System.err"` so that error output will go to the error stream (all console output for `java.util.logging` goes to stderr by default). See the API specs for Log4j and for J2SE for how to configure either product. If you are embedding SqlTool in a product to process SQL files, I suggest that you use log4j. `java.util.logging` is neither scalable nor well-designed.

Run the command `\l?` to see how to use the logging command `\l` in your SQL files (or interactively), including what logging levels you may specify.

Interactive Usage

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/sqltool.jar urlid
```

Procedure 1.3. 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 RC File 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

Any command may be preceded by space characters. Special Commands, Edit Buffer Commands, PL Commands, Macros always consist of just one line.

These rules do not apply at all to Raw Mode . Raw mode is for use by advanced users when they want to completely bypass SqlTool processing in order to enter a chunk of text for direct transmission to the database engine.

SqlTool Command-Line Editing

If you are really comfortable with grep, perl, or vim, you will instantly be an expert with SqlTool command-line editing. Due to limitations of Java I/O, we can't use up-arrow recall, which many people are used to from DosKey and Bash shell. If you don't know how to use regular expressions, and don't want to learn how to use them, then just forget command-recall. (Actually DosKey does work from vanilla Windows MSDOS console windows. Be aware that it suffers from the same 20-year old quirks as DOS command-line editing. Very often the command line history will get shifted and you won't be able to find the command you want to recall. Usually you can work around this by typing a comment... "::" to DOS or "--" to SqlTool then re-trying on the next command line).

Basic command entry (i.e., without regexps)

- Just type in your command, and use the backspace-key to fix mistakes on the same line.
- If you goof up a multi-line command, just hit the ENTER key twice to start over. (The command will be moved to the buffer where it will do no harm).
- Use the ":h" command to view your command history. You can use your terminal emulator scroll bar and copy and paste facility to repeat commands.
- As long as you don't need to change text that is already in a command, you can easily repeat commands from the history like ":14;" to re-run command number 14 from history.
- Expanding just a bit from the previous item, you can add on to a previous command by running a command like ":14a" (where the "a" means *append*).
- See the Macros section about how to set and use macros.

If you use regular expressions to search through your command history, or to modify commands, be aware that the command type of commands in history are fixed. You can search and modify the text after a \ or * prefix (if any), but you can't search on or change a prefix (or add or remove one).

Command Types

When you are typing into SqlTool, you are always typing part of the *immediate command*. If the immediate command is an SQL statement, it is executed as soon as SqlTool reads in the trailing (unquoted) semi-colon. Commands of the

other command types are executed as soon as you hit ENTER. The interactive : commands can perform actions with or on the edit buffer. The *edit buffer* usually contains a copy of the last command executed, and you can always view it with the :b command. If you never use any : commands, you can entirely ignore the edit buffer. If you want to repeat commands or edit previous commands, you will need to work with the edit buffer. The immediate command contains whatever (and exactly what) you type. The command history and edit buffer may contain any type of command other than comments and : commands (i.e., : commands and comments are just not copied to the history or to the edit buffer).

Hopefully an example will clarify the difference between the immediate command and the edit buffer. If you type in the edit buffer Substitution command "`:s/tbl/table/`", the :s command that you typed is the immediate command (and it will never be stored to the edit buffer or history, since it is a : command), but the purpose of the substitution command is to modify the contents of the edit buffer (perform a substitution on it)-- the goal being that after your substitutions you would execute the buffer with the "`:;`" command. The ":a" command is special in that when you hit ENTER to execute it, it copies the contents of the edit buffer to a new immediate command and leaves you in a state where you are *appending* to that *immediate* command (nearly) exactly as if you had just typed it in.

Emulating Non-Interactive mode

You can run SqlTool *interactively*, but have SqlTool behave exactly as if it were processing an SQL file (i.e., no command-line prompts, error-handling that defaults to fail-upon-error, etc.). Just specify "-" as the SQL file name in the command line. This is a good way to test what SqlTool will do when it encounters any specific command in an SQL file. See the Piping and shell scripting subsection of the Non-Interactive chapter for an example.

Command Types

Command types

SQL Statement

Any command that you enter which does not begin with "\", ":", "* " or "/" is an SQL Statement. The command is not terminated when you hit ENTER, like most OS shells. You terminate SQL Statements with either ";" 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 edit buffer and SQL command history for editing or viewing later on. In the former case, *execute against the SQL database* means to transmit the SQL text to the database engine for execution. In the latter case (you end an SQL Statement with a blank line), the command will go to the edit buffer and SQL history, but will not be executed (but you can execute it later from the edit 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, Edit Buffer / History Command, or PL Command, you are always *appending* lines to an SQL Statement or comment. (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 or comment).

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.

Edit Buffer / History Command

Run the command ":?" to list the Edit-Buffer/History Commands. All of these commands begin with ":". These commands use commands from the command

	history, or operate upon the edit "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 `* if` and `* while`. A few PL features (such as macros and updating and selecting data directly from/to files) can be a real convenience for nearly all users, so these features will be discussed briefly in this section. More detailed explanation of PL variables and the other PL features, with examples, are covered in the SqlTool Procedural Language section.
Macro Command	Macro definition and usage commands. Run the command "/?" to show the define, list, or use macros.
Raw Mode	The descriptions of command-types above do not apply to Raw Mode . In raw mode, SqlTool doesn't interpret what you type at all. It all just goes into the edit buffer which you can send to the database engine. Beginners can safely ignore raw mode. You will never encounter it unless you run the "\." special command, or define a stored procedure or function. See the Raw Mode section for the details.

Special Commands

Essential Special Commands

\?	In-program Help. **Run this to show ALL available Special Commands instead of just the subset listed here!**
\q	Quit
\j...	View JDBC Data Source details or connect up to a JDBC Data Source (replacing the current connection, if any). Run \? to see the syntax for the different usages.
\i path/to/script.sql	Execute the specified SQL script, then continue again interactively. Since SqlTool is a Java program, you can safely use forward slashes in your file paths, regardless of your operating system. You can use Java system properties like `${user.home}`, PL variables like `*{this}` and `@` in your file paths. The last is mostly useful for \i statements inside of SQL files, where it means the directory containing the *current* script.
\c true (or false)	Change error-handling (*Continue-on-error*) behavior from the default. By default when SqlTool is run interactively, errors will be reported but SqlTool will continue to process subsequent commands. By default when SqlTool is run non-interactively, errors will also cause SqlTool to stop processing the current stream (like stdin) or SQL file. The default settings are usually what is desired, except for SQL scripts which need to abort upon failures, even when invoked manually (including for interactive testing purposes).
\=	Commit the current SQL transaction. Most users are used to typing the SQL statement `commit;`, but this command is crucial for those databases which don't support the statement. It's obviously unnecessary if you have auto-commit mode on.

\m?	List a summary of DSV and CSV importing, and all available options for them. You can use variables in the file path specifications, as described for the \i command above.
\x?	Ditto.
\mq?	Ditto.
\xq?	Ditto.
\d?	List a summary of the \d commands below.
\dt [filter_substring]	
\dv [filter_substring]	
\ds [filter_substring]	
\di [table_name]	
\dS [filter_substring]	
\da [filter_substring]	
\dn [filter_substring]	
\du [filter_substring]	
\dr [filter_substring]	
\d* [filter_substring]	Lists available objects of the given type. • t: non-system Tables • v: Views • s: Sequences • i: Indexes • S: System tables • a: Aliases • n: schema Names • u: database Users • r: Roles • *: all table-like objects If your database supports schemas, then the schema name will also be listed. If you supply an optional *filter substring*, then only items which match the specified substring. will be listed. In most cases, the specified filter will be treated as a regular expression matched against the candidate object names. In order to take advantage of extreme server-side performance benefits, however,

in some cases the substring is passed to the database server and the filter will processed by the server.

The regexp test is case-sensitive!

Even though in SQL queries and for the "\d objectname" command object names are usually case-insensitive, for the \dX commands, you must capitalize the filter substring exactly as it will appear in the special command output. This is an inconvenience, since the database engine will change names in SQL to default case unless you double-quote the name, but that is server-side functionality which cannot (portably) be reproduced by SqlTool. You can use spaces and other special characters in the string.

Schema-narrowed Filter Specs

Filter substrings ending with "." are special. If a substring ends with ".", then this means to narrow the search by the exact, case-sensitive schema name given. For example, if I run "`\d* BLAINE.`", this will list all table-like database objects in the "BLAINE" schema. The capitalization of the schema must be exactly the same as how the schema name is listed by the "\dn" command. You can use spaces and other special characters in the string. (I.e., enter the name exactly how you would enter it inside of double-quotes in an SQL command). This is an inconvenience, since the database engine will change names in SQL to default case unless you double-quote the name, but that is server-side functionality which cannot (portably) be reproduced by SqlTool.

Current-Schema Filter Spec

The filter string "." (just a plain dot) means the current session schema, for databases which support the concept according to the SQL standard (HyperSQL database does).

Searching for Indexes

Indexes may not be searched for by *substring*, only by exact target table name. So if `I1` is an index on table `T1`, then you list this index by running "\di T1". In addition, many database vendors will report on indexes only if a target table is identified. Therefore, "\di" with no argument will fail if your database vendor does not support it.

\d objectname [[/]regexp]

Lists names of columns in the specified table or view. `objectname` may be a base table name or a schema.object name.

If you supply a filter string, then only columns with a name matching the given regular expression will be listd. (If no special characters are used, this just means that names containing the specified substring will match). You'll find this filter is a great convenience compared to other database utilities, where you have to list all columns of large tables when you are only interested in one of them.

To narrow the displayed information based on all column outputs, instead of just the column names, just prefix the expression with /. For example, to list all INTEGER columns, you could run `\d mytable /INTEGER`.

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.

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.

Edit Buffer / History Commands

Edit Buffer / History Commands

:?	IN-program Help
:b	List the current contents of the edit buffer.
:h	Shows the Command History. For each command which has been executed (up to the max history length), the SQL command history will show the command; its command number (#); and also how many commands *back* it is (as a negative number). : commands are never added to the history list. You can then use either form of the command identifier to recall a command to the edit buffer (the command described next) or as the target of any of the following : commands. This last is accomplished in a manner very similar to the vi editor. You specify the target command number between the colon and the command. As an example, if you gave the command `:s/X/Y/`, that would perform the substitution on the contents of the edit buffer; but if you gave the command `:-3 s/X/Y/`, that would perform the substitution on the command 3 back in the command history (and copy the output to the edit buffer). Also, just like vi, you can identify the command to recall by using a regular expression inside of slashes, like `:/blue/ s/X/Y/` to operate on the last command you ran which contains "blue".
:13 OR :-2 OR :/blue/	Recalls a command from Command history to the edit buffer. Enter ":" followed by the positive command number from Command history, like ":13"... or ":" followed by a negative number like ":-2" for two commands back in the Command history... or ":" followed by a regular expression inside slashes, like ":/blue/" to recall the last command which contains "blue". The specified

command will be written to the edit buffer so that you can execute it or edit it using the commands below.

As described under the :h command immediately above, you can follow the command number here with any of the commands below to perform the given operation on the specified command from history instead of on the edit buffer contents. So, for example, ":4;" would load command 4 from history then execute it (see the ":;" command below).

:; Executes the SQL, Special or PL statement in the edit buffer (by default). This is an extremely useful command. It's easy to remember because it consists of ":", meaning *Edit Buffer Command*, plus a line-terminating ";", (which generally means to execute an SQL statement, though in this case it will also execute a special or PL command).

:a Enter append mode with the contents of the edit buffer (by default) as the current command. When you hit ENTER, things will be nearly exactly the same as if you physically re-typed the command that is in the edit buffer. Whatever lines you type next will be appended to the immediate command. As always, you then have the choice of hitting ENTER to execute a Special or PL command, entering a blank line to store back to the edit buffer, or end a SQL statement with semi-colon and ENTER to execute it.

You can, optionally, put a string after the :a, in which case things will be exactly as just described except the additional text will also be appended to the new immediate command. If you put a string after the :a which ends with ;, then the resultant new immediate command will just be executed right away, as if you typed in and entered the entire thing.

If your edit buffer contains `SELECT x FROM mytab` and you run `a:le`, the resultant command will be `SELECT x FROM mytable`. If your edit 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.

You may notice that you can't use the left-arrow or backspace key to back up over the original text. This is due to Java and portability constraints. If you want to edit existing text, then you shouldn't use the Append command.

:s/from regex/to string/switches The Substitution Command is the primary method for SqlTool command editing-- it operates upon the current edit buffer by default. The "to string" and the "switches" are both optional (though the final "/" is not). 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.

In the example above, the from regex was a plain string, but it is interpreted as a regular expression so you can do all kinds of powerful

substitutions. See the `perlre` man page, or the java.util.regex.Pattern [http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html] API Spec for everything you need to know about extended regular expressions.

Don't end a *to* string with ";" in attempt to make a command 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 regex* case insensitive.
- Use "g" to substitute Globally, i.e., to substitute *all* occurrences of the *from regex* instead of only the first occurrence found.
- Use ";" to execute the command immediately after the substitution is performed.
- Use "m" for ^ and $ to match each line-break in a multi-line edit buffer, instead of just at the very beginning and every end of the entire buffer.

If you specify a command number (from the command history), you end up with a feature very reminiscent of vi, but even more powerful, since the Perl/Java regular expression are a superset of the vi regular expressions. As an example,

```
    :24 s/pin/needle/g;
```

would start with command number 24 from command history, substitute "needle" for all occurrences of "pin", then execute the result of that substitution (and this final statement will of course be copied to the edit buffer and to command history).

:w /path/to/file.sql

This appends the contents of the current buffer (by default) to the specified file. Since what is being written are Special, PL, or SQL commands, you are effectively creating an SQL script. To write some previous command to a file, just restore the command to the edit buffer with a command like ":-4" before you give the :w command.

I find the ":/regex/" and ":/regex/;" constructs particularly handy for every-day usage.

```
    :/\\d/;
```

re-executes the last \d command that you gave (The extra "\" is needed to escape the special meaning of "\" in regular expressions). It's great to be able to recall and execute the last "insert" command, for example, without needing to check the history or keep track of how many commands back it was. To re-execute the last insert command, just run ":/insert/;". If you want to be safe about it, do it in two steps to verify that you didn't accidentally recall some other command which happened to contain the string "insert", like

```
    :/insert/
    :;
```

(Executing the last only if you are satisfied when SqlTool reports what command it restored). Often, of course, you will want to change the command before re-executing, and that's when you combine the :s and :a commands.

We'll finish up with a couple fine points about Edit/Buffer commands. You generally can't use PL variables in Edit/Buffer commands, to eliminate possible ambiguities and complexities when modifying commands. The :w command

is an exception to this rule, since it can be useful to use variables to determine the output file, and this command does not do any "editing".

The :? in-program help explains how you can change the default regular expression matching behavior (case sensitivity, etc.), but you can always use syntax like "(?i)" inside of your regular expression, as described in the Java API spec for class `java.util.regex.Pattern [http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html]` . History-command-matching with the /regex/ construct is purposefully liberal, matching any portion of the command, case sensitive, etc., but you can still use the method just described to modify this behavior. In this case, you could use "(?-i)" at the beginning of your regular expression to be case-sensitive.

Command History

The SQL history shown by the :h command, and used by other commands, is truncated to 100 entries, since its 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 the desire integer value (using any of the System Property mechanisms provided by Java). If there is any demand, I'll make the setting of this value more convenient.

The SQL history list contains all executed commands other than Edit Buffer commands and comments, even if the command has a syntax error or fails upon execution. The reason for including bad commands is so that you can recall and fix them if you wish to. The same applies to the edit buffer. If you copy a command to the edit buffer by entering blank line, or if you edit the edit buffer, that edit buffer value will never make it into the command history until and if you execute it.

PL Commands

Essential PL Command

* ?	In-program Help about using the PL variables which have been set. **Use this command!**
* ? assign	In-program Help about setting and unsetting PL variables. **Use this command!**
* VARNAME = value	Set the value of a variable. If the variable doesn't exist yet, it will be created. The most common use for this is so that you can later use it in math expressions like `VARNAME`, in logical (conditionally) expressions like `*{VARNAME}`, or in other commands (including SQL) like `*{VARNAME}` or `*{:VARNAME}` construct. The only difference between `*{literal}` and `*{:VARNAME}` is that the former produces an error if VARNAME is not set, whereas the latter will expand to a zero-length string if VARNAME is not set.

Preventing unset-variable Errors

You can prevent all unset-variable errors by using the construct `*{:VARNAME}` in place of `*{VARNAME}` wherever `VARNAME` may not then be set.

Warning

You can use this assignment command to set the value of a variable to the empty string (unless you set Java system property `sqltool.REMOVE_EMPTY_VARS` to `true`). We are talking about assignments like the following:

```
    * VARNAME =
```

Regardless of `sqltool.REMOVE_EMPTY_VARS`, you can always use the unset command (described next) to unset variables, and you can always use command line switches `-P` or `--setVar` to assign empty strings.

See Variables subsection for information about variable usage.

* - VARNAME	*Unset* (remove) the specified variable.
* VARNAME _	When next SQL command is run, instead of displaying the rows, just store the very first column value to variable VARNAME. This works for CLOB columns too. It also works with Oracle XML type columns if you use column labels and the `getclobval` function. If the SQL null value is retrieved next, then this variable will be assigned the value null, which is the same thing as unsetting it. It's easy to tell when a variable is set to null vs. when it is set to the empty string. See the Nulls and Empty Strings section about that.
* ? control	In-program Help about PL control/branching commands. **Use this command!**
* if (LOGICAL EXPR)	If the logical expression evaluates to true, then the following block of code (up to the paired `* end if` statement is executed. If the expression is false, then the same code block is skipped. Run `* ? control` for details, including the optional `* else` statement, a short-cut *inline if statement*, and several other branching statements.

This list here includes only a sampling of some *essential* PL Commands, but **there are many other useful PL Commands which you can list by running `* ?`** .

PL variables are intimately involved with most PL commands, and (and with some Special commands). Even if you never assign a PL variable, if you are at technical level of using PL commands, you should at least know how to check SqlTool system PL variables which effect SqlTool's behavior. See the Nulls and Empty Strings section about that.

Non-Interactive

Read the Interactive Usage section if you have not already, because much of what is in this section builds upon that. You can skip all discussion about Command History and the edit buffer if you will not use those interactive features. (Except the important exception that the edit buffer is still populated by executed commands and raw mode, so the buffer can be used by `* VARNAME :`, `/: VARNAME`, `\x :`, and `\xq :` commands).

The previous point brings us to another important consideration for SQL script writers. When SqlTool is run interactively, you can enter a blank line after a SQL command to send the command to the edit buffer without executing it. That action is not supported in scripts, however, because scripters expect more freedom in usage of white space. I.e., scripters should be able to add blank lines wherever they want to in their scripts-- and they can. The problem is, defining variables or macros or performing exports using multi-line SQL statements requires the multi-line SQL statements in the edit buffer. One way to do these commands into the buffer is to execute the SQL command, but usually you do not want the SQL to execute until expansion or execution time of the variable/macro/export. The *empty-line* method only works in interactive mode. What we use is Raw Mode . This works great both interactively and non-interactively, and it supports Chunking without having to format your SQL in a special way. A great application of this is to put multi-line macro and function definitions into your `auto.sql` file.

SqlTool system PL variables control behavior (for example, they control many aspects of DSV importing and exporting). User PL variables can be used to make your scripts dynamic and for conditional actions. Both system and user PL variables can be set by `--setVar` and `-p` switches, or PL commands in `--sql` switches or SQL files (as well as in `auto.sql` for interactive usage). Since the variables are all global and shared across contexts, the variables thus set effect behavior of all subsequence content in `--sql` switches and SQL files (and `auto.sql` and stdin for interactive usage). See the Variables subsection for the particulars.

Remember to Commit

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 detect errors and to control JDBC transactions. SqlTool is designed so that you can reliably determine if errors occurred within SQL scripts themselves, and from the invoking environment (for example, from a Perl, Bash, or Python script, or a simple cron tab invocation).

Giving SQL on the Command Line

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

```
java -jar $HSQLDB_HOME/lib/sqltool.jar --sql="SQL statement;" urlid
```

Note

The `--sql` switch automatically implies `--noinput`, so if you want to execute the specified SQL before *and in addition to* an interactive session (or stdin piping), then you must also give the *--stdinput* switch.

Since SqlTool transmits SQL statements to the database engine only when a line is terminated with ";", if you want feedback from multiple SQL statements in an --sql expression, you will need to use functionality of your OS shell to include linebreaks after the semicolons in the expression. With any Bourne-compatible shell, you can include linebreaks in the SQL statements like this.

```
    java -jar $HSQLDB_HOME/lib/sqltool.jar --sql='
        SQL statement number one;
        * NEWVAR = something
        \p A SqlTool Special command which echoes NEWVAR: *{NEWVAR}
            number two;
        SQL statement three;
    ' urlid
```

Notice that the *SQL string* is not strictly SQL, but SqlTool input, so it may contain Special or PL commands. The variable is set this way only for educational purposes. The same thing could be accomplished more elegantly by using the `-p` switch.

Note

The multi-line examples in this section will only work as-is with a Bourne-compatible shell. With some ugliness they can be converted to C shell. For Windows, you are better off to stick with SQL files for multi-line input.

If you don't need feedback, just separate the SQL commands with semicolons and the entire expression will be chunked .

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/sqltool.jar --sql='
        select count(*) from usertbl;
    ' urlid` || {
        # Handle the SqlTool error
    }
```

```
    echo "There are $USERCOUNT users registered in the database."
    [ "$USECOUNT" -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/sqltool.jar urlid file.sql... > /tmp/file.log 2>&1
```

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 `\c 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 `\c false` to the top of the script.

If you specify multiple SQL files on the command-line, the default behavior is to exit SqlTool immediately 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 contains error handling statements, they will take precedence.

You can also use \i in SQL files to pull in (nest) additional SQL files. This is a powerful way to hierarchically maintain and configuration manage a set of scripts for a product, project, or database. For encapsulation, tracking, and collaboration purposes, it's usually best to keep each SQL script focused on one task or goal, for example: creating a table, it's trigger, and loading initial data from a DSV file. Usually a set of such scripts will have to be executed in a precise order so that referenced tables are created before tables with foreign keys to them, etc. I make a *super-script* for every database project that I manage. Besides this strategy proving and configuration managing an installation procedure known to work, I can recreate large and complex, custom product databases for deployments or tests in seconds. With the addition of some very simple PL coding, I can re-create all database structures in a parallel schema by just specifying a schema name when I invoke the super-script.

Only for interactive SqlTool invocations, the file `auto.sql` in your home directory will be executed before your typed (or piped) input. It is processed in the same way as a script file specified on the command-line except that since you are running SqlTool interactively, the interactive SqlTool rules will apply. If your `auto.sql` does setup that you need done for non-interactive SQL files, then add a `\i ${user.home}/auto.sql` to the top of the script, understanding that if you execute that script interactively you will cause `auto.sql` to be executed a second time, but see the following tip about preventing that.

Preventing redundant execution of utility or shared scripts.

There's a common idiom used in UNIX login and shell initialization scripts that prevents redundant executions, and it works great for SqlTool too. Think up a unique PL variable name which will keep track of whether the script has been sourced in the current SqlTool session, and the script will have no effect if called a 2nd, 3rd, etc. time. This example applies the idiom to an `auto.sql` file, but it can be used in any script that you want to prevent superfluous executions in a developer-friendly way.

```
    * if (*AUTO_EXECUTED)
        * break
    * end if
    * AUTO_EXECUTED = true
```

It would be slightly more efficient, but less reliable, to put the test on the caller's side instead of the callee side, like this:

```
    * if (! *AUTO_EXECUTED)
        * \i ${user.home}/auto.sql
    * end if
```

As you would probably guess, all of SQL's file commands (for example loading or saving SQL scripts, binary data, DSV data) take either relative or absolute file paths. However, when you nest scripts, you will usually want to begin your paths with the `@/`. The initial `@` character in file paths means *the directory containing the current script*. This is important because of Java's frustrating inability to switch the current directory. By using `@`, you can cross-reference between SQL scripts in one or several co-located directories, and everything will *just work*, regardless of your current directory when you invoke script(s).

Make use of @/ construct for CMD scripts with nesting

If you use a managed set of nested scripts, you are advised to prefix all relative file paths inside of SqlTool scripts with `@/`. Without this, relative file paths would be resolved relative to the invocation directory--making your scripts require a specific invocation directory. The *at character* will resolve to the directory containing the current script.

You can use the following SQL file, `sample/sample.sql` , from your HyperSQL distribution [1]. It contains SQL as well as Special Commands making good use of most of the Special Commands documented below.

```
/*
    $Id: sample.sql 3637 2010-06-07 00:59:13Z unsaved $
    Exemplifies use of SqlTool.
    PCTASK Table creation
*/

/* Ignore error for these two statements */
\c true
DROP TABLE pctasklist;
DROP TABLE pctask;
\c false

\p Creating table pctask
CREATE TABLE pctask (
    id integer identity,
    name varchar(40),
    description varchar(256),
    url varchar(80),
    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 boolean 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/sqltool.jar  --sql='
        create user tomcat password "x";
    ' mem path/to/hsqldb/sample/sample.sql
```

This shows how you can mix SQL on the command line, and SQL inside an SQL file.

Note

The example above uses Bourne shell syntax. C shell syntax would be similar. You would need to use an SQL file to accomplish this on Windows.

(The `--sql="create...;"` argument in the example creates an account which the following script uses). You should see error messages between the `Continue-on-error...true` and `Continue-on-error...false`. The script purposefully runs commands that might fail there. The reason the script does this is to perform database-independent conditional table removals. (The SQL clause `IF EXISTS` is more graceful and succinct, so you may want to use that if you don't need to support databases which don't support `IF EXISTS`). If an error occurs when continue-on-error is false, the script would abort immediately.

Tip

It can be very convenient to end-users to write SQL scripts that take a parameter(s) but use default values if none is specified.

Example 1.2. Default parameter value with optional user-specified override

```
# User can specify a value like...
java -jar .../sqltool.jar -Pvarname=override urlid script.sql
# Or use the default varname value:
java -jar .../sqltool.jar urlid script.sql
```

```
-- In your script assign your default value like so:
*if (*varname == **NUL) * varname = default value
```

This works in "auto.sql" file too.

Piping and shell scripting

You can of course, redirect output *from* SqlTool to a file or another program.

```
java -jar $HSQLDB_HOME/lib/sqltool.jar urlid file.sql > file.txt 2>&1

java -jar $HSQLDB_HOME/lib/sqltool.jar urlid file.sql 2>&1 | someprogram...
```

You can type commands in to SqlTool while being in non-interactive mode by supplying "-" as the file name. This is a good way to test how SqlTool will behave when processing your SQL files.

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

This is how you have SqlTool read its input from another program:

Example 1.3. Piping input into SqlTool

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

Example 1.4. Redirecting input into SqlTool

```
    java -jar $HSQLDB_HOME/lib/sqltool.jar urlid - < myFile.sql
```

For a shell not as graceful as the Bourne-compatible shells, you would need to type this all on the same line (or use a line-continuation trick).

Warning

Beware of null stdin to SqlTool (or SqlFile). At least with Java 6 on UNIX, `System.in` returns megabytes of garbage for reads if stdin is closed. I consider this an obvious bug. Therefore, unlike any other program you would invoke from scripts, check stdin before running any Java program that will read from it. I consider this a big ugly bug in Java. This is not just theoretical, because many remote execution environments will have stdin closed off.

Make sure that you also read the Giving SQL on the Command Line section. The `--sql` and `-p` switches are great facilities to use with shell scripts.

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 \c 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 SqlTool Procedural Language section.

Tip

Since SqlTool religiously returns meaningful exit status, you can use the following idiom to send alerts about failed batch jobs (for example, jobs started by cron, at, AutoSys, Quartz, Hudson).

Example 1.5. Error-handling Idiom

```
java -jar $HSQLDB_HOME/lib/sqltool.jar urlid $HOME/app/myFile.sql >> $HOME/log/app.log
 2>&1 ||
echo "See log file y:$HOME/log/app.log" | mailx -s "App aborted on host y"
 recip1@z.com recip2@z.com
```

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=true

  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

Comments of the form `/*...*/` or `--` behave as a SQL programmer would expect, in all contexts other than in interactive edit/history commands.

If a comment occurs outside of an SQL statement, SqlTool will not send the comment to the database (to improve performance). Raw mode can be used to send just comments to the database. In order to proactively catch accidents, SqlTool will complain if you attempt to send an empty SQL statement (i.e., just whitespace) to the database, even in raw mode.

Special Commands and Edit Buffer Commands in SQL Files

Generally, don't use Edit Buffer / History Commands in your sql files, because they won't work. Edit Buffer / History Commands are for interactive use only. However, the important scripting commands `* VARNAME :`, `/: VARNAME`, `\x :`, and `\xq :`, as well as Raw Mode do use the edit buffer.

You can, of course, use any SqlTool command at all interactively. The goal here is just to group together the commands most useful to script-writers.

\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 .../sqltool.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 SqlTool 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. As a result,

```
\q
```

means to make an immediate but successful exit, whereas

```
\q Message
```

means to abort immediately with error status. Both commands will exit gracefully.

\p [text to print]	Print the given string to stdout. Just give "\p" alone to print a blank line.
\l SEVERITY_LEVEL text to log	The logging subsystem will display and/or log and/or email or whatever, depending on how you have it configured. To see where messages go by default, just play with it interactively. Run "`\l?`" to list the available severity levels.
\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.
\o [file/path.txt]	Tee output to the specified file (or stop doing so). See the Generating Text or HTML Reports section.
\=	A database-independent way to commit your SQL session. Useful for database which have no `COMMIT` SQL statement.
\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.
\c [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 \c 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` 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 `\c 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 \c 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.

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/sqltool.jar urlid < filepath1.sql > /tmp/log.html 2>&1
```

or

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

For a shell not as graceful as the Bourne-compatible shells, you would need to type this all on the same line (or use a line-continuation trick).

Character Encoding

There are several levels of encoding settings. First there are your platform defaults. These can be changed, temporarily or permanently, with system settings or environmental variables. Java system properties may be used to change the encodings for the JVM run. Finally, can specify a different encoding in your RC file, as documented in the RC File Authentication Setup section, though these will not effect stdin or stdout (as explained there). Programmatic users of `SqlFile` have complete control over encoding by setting up `Reader`s and `PrintWriter`s, or by using constructors with an `encoding` parameter. Developers should understand that where a `SqlFile` constructor takes a `Reader` or a `PrintWriter` parameter, we will not apply encoding settings to them, leaving that up to you.

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.

Note

HTML reporting has been drastically modernized. It now has user-overridable boilerplates, flexible and safe CSS styles, and PL variable substitution.

Procedure 1.4. 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 true`. 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. You must set HTML mode to true before running the `\o` command of the next step, because this is how the `\o` command knows to write the opening HTML (header and such) to the file.

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). If you are in HTML mode and you are finished writing the file (i.e. you will not append to it again later), then close it with `\oc` instead, to

4. If you turned HTML mode on before and want to turn it off now, run `\h false`.

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 pre-existing target file ahead of time.

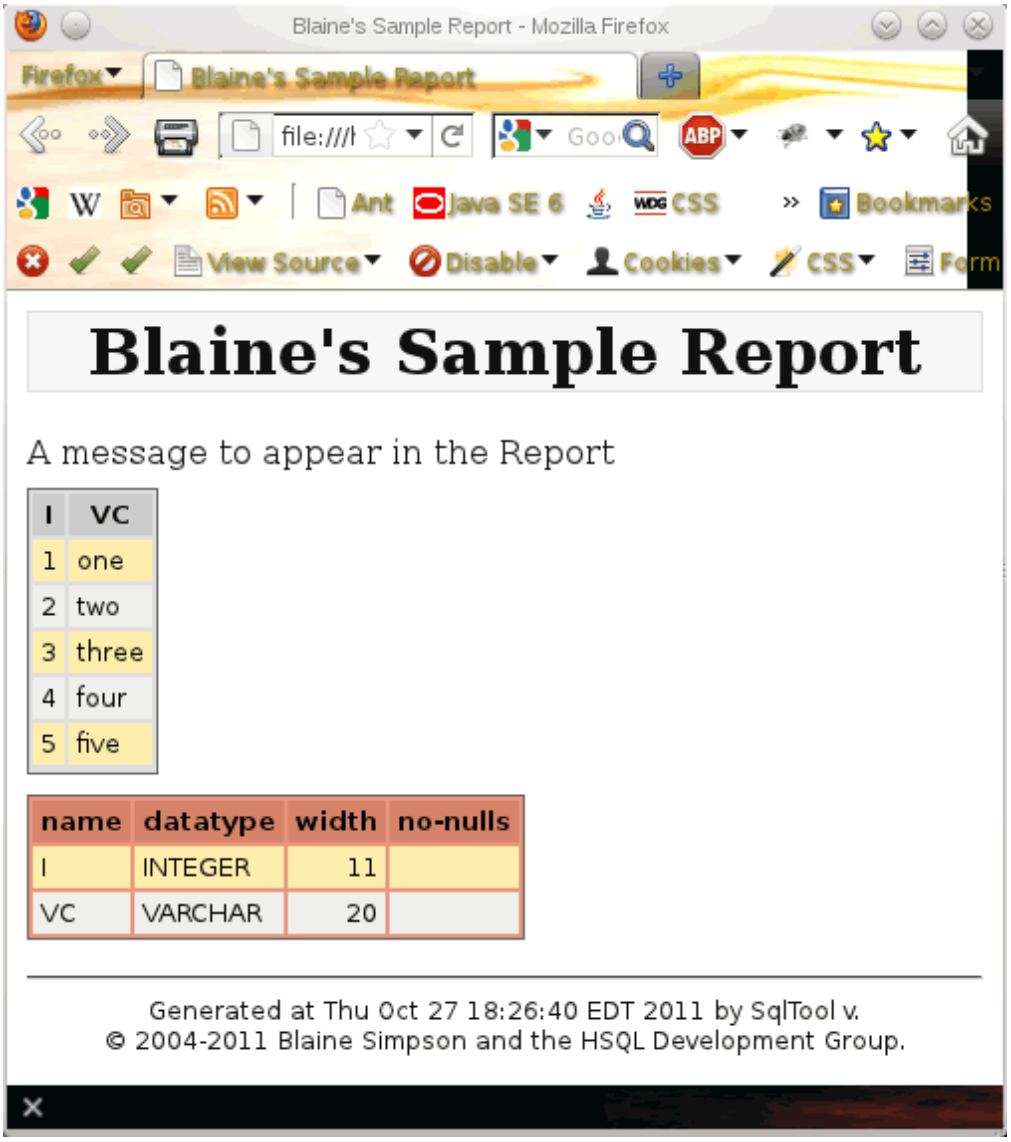
Tip

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 ":/select/;" to repeat the last SQL command containing the given string ("select" in this example). 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 ":;" Edit Buffer Command.

HTML output has its own `*NULL_REP_HTML` setting distinct from `*NULL_REP_TOKEN`. It fulfils `*NULL_REP_TOKEN`'s output purpose of saying how to represent SQL nulls retrieved from VARCHAR columns, but lets you manage it for HTML separately from display and CSV/DSV output.

Several new features have been added to HTML reporting between revisions 4505 and 4606 inclusive. All of the new features are explained in comments in the working, sample SQL file below. If you write your own top boilerplate fragment, you will probably want to style the CSS classes written by SqlTool. These class names all begin with `sqltool-`, to avoid namespace collisions. Just generate a sample report to see which what class names get used.



A HTML report

The report above has a simple table dump and the basic column definitions for that table. You can, of course, make reports with any number of queries of any level of sophistication. The SQL file below generated this report.

Please study this example closely, because this is your principal source of education about the specifics of creating HTML reports. Notice the close at the very end with the explicit HTML-close command `\oc`. If you used just `\o`, the file would be closed but the closing HTML code would not be written. `csv-sample.sql` [1].

Example 1.6. Sample HTML Report Generation Script

```
/*
 * $Id: html-report.sql 4564 2011-10-19 04:27:37Z unsaved $
 *
 * Sample/Template for writing an HTML Report
 */

-- Populate sample data
create table t (i integer, vc varchar(20));
insert into t values(1, 'one');
```

```
insert into t values(2, 'two');
insert into t values(3, 'three');
insert into t values(4, 'four');
insert into t values(5, 'five');
commit;


-- IMPORTANT:  \o will append by default.  If you want to write a new file,
-- it's your responsibility to check that a file of the same name does not
-- already exist (or remove it).


-- Follow the following examples to use your own HTML fragment files.
-- * *TOP_HTMLFRAG_FILE = /tmp/top.html
-- * *BOTTOM_HTMLFRAG_FILE = /tmp/bottom.html

-- The default TOP_HTMLFRAG_FILE has a reference to this PL variable.
* REPORT_TITLE = Blaine's Sample Report
-- The default will also override its CSS style settings with your own if you
-- put them in a file named "overrides.css" in same directory alongside your
-- reports ("report.html" in this example).
-- You can add references to ${system.properties} and *{PL_VARIABLES} in
-- your own custom fragment files too.


-- Turn on HTML output mode.
-- Must enable HTML _before_ opening to write top frag.
\h true
\o report.html
\p A message to appear in the Report
SELECT * FROM t;

-- Close off output just to show that you can go back and forth.
-- A close with '\o' will not write the bottom boilerplate that closes the HTML.
\o
\h false
\p Some non-HTML non-Report output:
SELECT count(*) FROM t;

\h true
-- Re-open the report
\o report.html
\d t
-- This time close it with
\oc
```

One thing that I chose not to exemplify in the example, so as not to scare away less technical users, is that you can use the `\p` variant command `\pr`. When you give a `\p` command in HTML mode, SqlTool formats the output into a paragraph. But coders may want to write HTML, Javascript, JSP, or similar code, and SqlTool should treat this as *Raw* to be written as-is to the report file. This is what `\p` accomplishes.

Storing and Retrieving Binary Files

You can upload binary files such as photographs, audio files, or serialized Java objects into database columns. SqlTool keeps one binary buffer which you can load from files with the \bl command, or from a database query by doing a one-row query for any non-displayable type (including `BLOB`, `OBJECT`, and `OTHER`). In the latter case, the data returned for the first non-displayable column of the first result row will be stored into the binary buffer.

Once you have data in the binary buffer, you can upload it to a database column (including `BLOB`, `OBJECT`, and `OTHER` type columns), or save it to a file. The former is accomplished by the special command \bp followed by a *prepared* SQL query containing one question mark place-holder to indicate where the data gets inserted. The latter is accomplished with the \bd command.

You can also store the output from normal, displayable column into the binary buffer by using the special command \b. The very first column value from the first result row of the next SQL command will be stored to the binary byte buffer.

Example 1.7. Inserting binary data into database from a file

```
    \bl /tmp/favoritesong.mp3
    \bp
    INSERT INTO musictbl (id, stream) VALUES(3112, ?);
```

Example 1.8. Downloading binary data from database to a file

```
    SELECT stream FROM musictbl WHERE id = 3112;
    \bd /tmp/favoritesong.mp3
```

You can also store and retrieve text column values to/from ASCII files, as documented in the Essential PL Command section.

SqlTool Procedural Language

Aka PL

Most importantly, run `SqlTool` interactively and give the "`*?`" command to see what PL commands are available to you. I've tried to design the language features to be intuitive. Readers experience with significant shell scripting in any language can probably learn the rudiments by looking at (and running!) the sample script `sample/pl.sql` in your HyperSQL distribution [1] and then pick up everything else by using the `*?` command from within an interactive SqlTool session. (By *significant* shell scripting, I mean to the extent of using variables, for loops, etc.).

It generally causes an error to reference a variable that has not been set. SqlTool will always attempt to de-reference PL variable and Java system property references, except for (a) in : commands, and (b) in SQL statements if no user PL variable has been set. Since you should never be trying to dereference a PL variable if none have been set, the practical implications are just: If you want Java system properties to be de-referenced, just make sure that any PL user variable is set; and if you have strings like `${this}` appear in your SQL text (which you do not want expanded), unset all user PL variables before executing that text (if any have been set earlier). The purpose of this system is to avoid changing user-specified SQL without the user knowing it. People who don't use PL at all don't have to worry about strings getting accidentally expanded.

PL is also used to upload and download column values to/from local ASCII files, analogously to the special \b commands for binary files. This is explained above in the Interactive Essential PL Command section above.

Nulls and Empty Strings

I am raising this PL variable topic out of order here because it is important to understand, and I'd like you to have the concept firmly in mind before digging into other details about variables. In this sub-section I am only talking about PL variable values (not system properties or SQL engine variables, etc.).

Very similarly to Java system properties, if the value of a PL variable is null, then the variable is *unset*; and (if you set the `sqltool.REMOVE_EMPTY_VARS` mode as suggested) there is no way to directly assign a variable to null. If, for example, you ran

```
            * MYVAR = null
```

that would assign the string value of `null` to your variable, not the real null value. If a variable is assigned null indirectly, say by fetching a null cell value into a variable, or when variable `?` is assigned null due to a SQL failure, this action is entirely equivalent to unsetting the variable.

Recap

If a variable is unset, or has never yet been set, then that variable's value is null. If a variable's value is null, then the variable is unset.

You can assign a PL variable the empty string value by using a command switch `-P` or `--setVar`. For example

```
java -jar .../sqltool.jar -Pvarname= urlid script.sql
```

You can see it's an empty string by echoing the value:

```
\p One*{varname}Two
OneTwo
```

With the next minor release of SqlTool, or now if you set Java system property `sqltool.REMOVE_EMPTY_VARS` to `false`, that you can use the vanilla PL assignment command to assign empty values like this:

```
* MYVAR =
```

Distinguishing Nulls from Empty Strings

You can't use the simplest Special Command `\p` to distinguish null variables, because (a) It is an error to expand an unset/null value with the `${x}` construct, and the alternative `${:x}` is safe but displays null values as if they were empty strings-- thereby not distinguishing. But don't fear. There are easy ways.

Unset/null variables are not listed by the commands `* list` or `* listvalues`. Therefore, say you want to know if variable `MYVAR` is set or not (stated differently, whether it is non-null or null). Run

```
    * list MYVAR
```

. If it lists the variable then it is set and is not null.

You can use the `* if` command to compare your variable to the null value or to the empty string. The first test here will tell you if `MYVAR` is equal to null (by comparing it to reserved PL variable `*NULL`). The second test here compares `MYVAR` to a variable that you assigned the empty string to earlier.

Example 1.9. Explicit null and empty-string Tests

```
    java -jar .../sqltool.jar -pEMPTYSTRING=
    ....
    * if (*MYVAR == **NULL)
        \p MYVAR really is null
    * end if
    * if (*MYVAR == EMPTYSTING)
        \p MYVAR is now an empty string
    * end if
```

There's an example in this chapter showing how to leverage this feature to set default values for optional user-specified parameters.

Definitely study the Special values for ?, and _ (or ~) Variables example.

Variables

Following subsections explain important things about specific variable types. Here we just list the variable types and give a few points about variable usage generally.

Variable Types

Database/SQL Variables	SqlTool has no control over variable mechanisms provided by the SQL implementation or database vendor. You can use such constructs only in SQL commands, since the other command types never reach the database engine. Nothing else that we have to say about manuals applies to database/SQL variables.
Java System Properties	SqlTool allows for reading but not writing of these variables with `${varname}` and `* listsysprops`. To prevent your SQL text from being changed unintentionally, `${varname}` occurrences will not be expanded inside of SQL statements unless at least one PL user variable has been set. Therefore, if executing portable SQL scripts (and by default), SqlTool will not expand `${varname}`s inside of SQL statements.
PL User Variables	These variables have names beginning with a letter and (if the name is longer than one character) any number of letter, digit, or _ characters. The letters are case-specific. Two examples are `m` and `MY_VAR`. There variables are created and assigned values on the SqlTool command-line or with any of several PL assignment commands listed by the `* ?` command. Depending on context (see below about that), they are referenced as `*{MY_VAR}`, as `*MY_VAR`, or as `MY_VAR`. You can display all current user (and SqlTool system) variables with the PL command `* listvalues`.
SqlTool System PL Variables	These are PL variables just like PL user variables, but the variable names begin with the `*` character, like `*DSV_TARGET_FILE`, and they effect SqlTool system behavior. Some of these are initialized by SqlTool automatically. You can change and examine the values in the same way as PL user variables. See the following subsection about System PL Variables for details.
Reserved PL Variable ?	The ? variable is set automatically to the results of SQL statement executions. The reset state is the empty string, and it is only ever set to null (aka *unset*) if an SQL error was encountered.
*PL variables NULL and *NULL	These are actually reserved system and user PL variables, and since they are very unique and interchangeable with one another, I'm giving them their own bullet. These are reserved PL variables which always have the value of null (which has the meaning of *unset*. You can compare other variables to `*NULL` or `NULL` to see if they are set or not. A specific application is to compare *?* to **NULL* or *NULL* to see if the last SQL command has failed.

General Rules for PL vars and Java system props

- PL variables and Java system properties are always expanded in Special, and (most) PL commands if they are written like `*{VARNAME}` and `${VARNAME}` correspondingly. They are expanded in the same way inside of SQL statements as long as one (or more) PL variable has been set.
- Your SQL scripts can give good feedback by displaying the value of variables with the "\p" Special command.

System PL Variables

Important

Definitely run command `* ?` to view in-program help about referencing PL variables, and run `* ? assign` if you will be assigning variables.

SqlTool automatically assign values to a few special system variables. As I write this, the special variables are only `*START_TIME`, `*REVISION`, `*TIMESTAMP`. `*START_TIME` is a date and time string formatted for the user's locale. `*REVISION` is SqlTool's version string (i.e., it is not a valid real or integer number). `*TIMESTAMP` is a user-configurable date or time string configurable with another system variable `*TIMESTAMP_FORMAT`.

SqlTool System PL variables are the mechanism used to configure SqlTool behavior. You can list all *set* PL variables by running the SqlTool command `* listvalues`. If a SqlTool System variable is not shown, then it is *unset* (which is equivalent to *non-null*). But if a system variable is not set, that doesn't mean that the setting behavior will be *unset*, but rather that the *default behavior* will apply. For example, if you `* listvalues` and the variable `*DSV_COL_DELIM` is not listed, that doesn't mean that there will be no DSV column delimiter, but that the default DSV column delimiter will be used. The in-program help can be used to determine what the default behavior is. (In the case of `*DSV_COL_DELIM`, you can see the default behavior by running `\x?`.

See the list of system variables in the SqlTool System PL Variables appendix.

PL Variables

This subsection explains points common to most or all of the PL variable varieties (all variables other than Database/SQL and Java system properties).

The new `-p` switch is an easy and elegant way to set PL variables when you know the needed values at SqlTool invocation time. This is a more user-friendly variant of the `--setVar` switch. The primary benefit is that you can specify multiple variable assignments using multiple `-p` switches, eliminating the need to separate `name=value` elements with commas (doing this necessitates the usage of `\,` escapes when there are commas in your intended variable values). The most basic usage is like `-PNAME=value`, but there are a few things to know to make this feature more useful. Firstly, the space after `-p` is optional, so you can write either `-PNAME=value` or `-P NAME=value`. Secondly, the 'p' itself is case-insensitive. You may choose to always user upper-case or always lower-case to be consistent. But if you do not put space after the p, I recommend that you change the capitalization of the p to more easily distinguish your variable names, like `-pVARNAME=x` and `-Pvarname=y`.

- PL variables are global to a SqlTool invocation and are therefore shared among
  - `--setVar` command-line switches.
  - `-P` or `-p` command-line switches.
  - `--sql` command-line switches.
  - `auto.sql` file, if it is present and the rules call for it to load.
  - SQL files loaded with \i from top or a nested level.
  - standard input whether from a terminal, redirection, or piping

  The variable must, of course, be set at a point in time before it is referenced.
- Use the `* list` command to list some or all variables; or `* listvalues` to also see the values. (Exception: The *EXCEPTION variable can not be displayed with the list commands).
- **Assignment**

A Mnemonic

The mnemonic distinction between assignment commands `* VARNAME _` and `* VARNAME ~` is that the latter shows the output, which you can think of as looking like ~ on your computer display. See the in-program help (`*?`) about the purpose and usage of these two commands.

Run the `* ?` command to see a list of commands that you can use to assign and to unset PL variables. The most simple assignment command is `* VARNAME = Var value`, but you can assign values from command output, query return values, contents of files, mathematical expressions, the edit buffer, etc.

Only the `* VARNAME :` assignment variant supports assigning a multi-line SQL statement(s) as body. To populate the edit buffer with your multi-line SQL query for the : assignment, you must execute the SQL command before (usually undesirable), or end the SQL with a blank line instead of a `;` only works interactively), or use Raw Mode .

- You can also set PL variables other than ? using the `--setVar` and `-P` (also usable as `-p`) command-line switches. I give a very brief but useful example of this below.
- You can unset (remove) PL variables using the `* - VARNAME` command.
- It is an error in a Special and most PL commands to expand an *unset* (remove) variable with `*{VARNAME}` or `${VARNAME}`. Therefore, if the variable/property may not be set, just add a colon like `*{:VARNAME}` or `${:VARNAME}` to expand the variable if set, but expand to a zero-length string if the variable is not set.
- Inside of logical expressions (like inside of `if` and `while` commands), reference variables like `*VARNAME`, i.e. without the curly brace, and don't worry about a construct like `${:VARNAME}` because it is legal to compare unset variables (all unset variables are equal to one another). The justfication for this simplification is explained below.
- The assignee variable name, and variables inside of mathematical expressions are written simply as bare words. For example: "`* VARNAME = Var value` (`*` there is a command prefix-- not part of the variable specifier) and "`* ((VARNAME = OTHER_VARNAME * 6))`".

PL commands can be used to upload and download column values to/from local ASCII files, but the corresponding actions for binary files use the special \b commands. This is because PL variables are used for ASCII values and you can store any number of column values in PL variables. This is not true for binary column values. The \b commands work with a single binary byte buffer.

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

PL ? Variable

You don't set the ? variable. It is much like the Bourne shell variable $? in that it is always automatically set to the first value of a result set (or the return value of other SQL commands). It works very similarly to the `* VARNAME ~` and `* VARNAME ~` assignment commands, but the value of ? is set automatically without you doing anything. You can, of course, dereference ? like any PL variable or view it with `* list` or `* listvalues`. If you are running interactively or have turned on `\c` (continue-upon-error), you should be prepared that `?` could get unset by SQL failures and thereby cause `*{?}` references to fail. (In which case the list commands still work, you can check it with an `* if` comparison, and the `*{:?}` construct will be safe (though this last does not show you the difference between empty string and null). The important thing to remember about the list commands is that variables that are not listed are *unset* (i.e., are null).

`?` is reliably set to null only upon SQL failures. Upon SqlTool startup, `?` is set to the empty string "" instead of being unset or null. If a query returns a null value in the last cell, then `?` will be assigned to the current `*DSV_NULL_REP` value instead of the literal null value. Therefore if you enable continue-on-error with `\c true` (or in interactive mode when this is the default... though I can't think of how this could be useful interactively), you can test for SQL failures with

```
    * if (*? == *NULL)
```

(`*NULL` is a reserved PL variable that always has the value of null, which means *unset*).

The important functional difference between variables assigned with `VARNAME _` or `VARNAME ~` vs. `?` is that the latter is always set to the last SQL cell value fetched (or return value for non-result-set SQL). Explicit assignments with _ or ~ are made from the very next cell content retrieved after the _ or ~ command (or return value for non-result-set SQL). Easier to show what I mean than to explain it...

Example 1.10. Special values for ?, and _ (or ~) Variables

```
sql> \p At startup ? is equal to empty string.  See between A and B:  A*{?}B
At startup ? is equal to empty string.  See between A and B:  AB
sql> * if (A*{?}B == AB) \p ? is the empty string
? is the empty string
sql>
sql> CREATE TABLE t(i INTEGER, vc VARCHAR(20));
sql> INSERT INTO t VALUES(1, 'one');
1 row updated.
sql> INSERT INTO t VALUES(2, 'two');
1 row updated.
sql> * res ~
sql> SELECT * FROM t;
I  VC
-  ---
1  one
2  two

Fetched 2 rows.
sql> \p *{?}
two
sql> \p *{res}
1
sql> * listvalues ? res
Listing all 'set' variables (any var not seen is unset and equal to null).
The outermost parentheses are not part of the values.
    ?: (two)
    res: (1)
sql>
sql> INSERT INTO t VALUES (3, null);
1 row updated.
sql> *res ~
sql> SELECT vc FROM t WHERE i = 3;
[null]
sql> \p *{?}
[null]
sql> * if (*res == **NULL) \p res really is null
res really is null
sql> * listvalues ? res
Listing all 'set' variables (any var not seen is unset and equal to null).
The outermost parentheses are not part of the values.
    ?: ([null])
sql>
sql> -- This will prevent SqlTool from aborting when we run a bad SQL statement:
sql> \c true
Continue-on-error is set to true.
sql> *res ~
sql> SELECT hocus FROM pocus;
SEVERE  SQL Error at '<stdin>' line 23:
"SELECT hocus FROM pocus"
user lacks privilege or object not found: POCUS
sql> * if (*? == **NULL) \p ? really is null
? really is null
SEVERE  Did not finish setting variable 'res' before a code block exited.
SEVERE  Rolling back SQL transaction.
sql> * if (*res == **NULL) \p res really is null
res really is null
sql> * listvalues ? res
Listing all 'set' variables (any var not seen is unset and equal to null).
The outermost parentheses are not part of the values.
```

```
sql>
```

(The SQL that generated this is available in the file `nullempty.sql` in the `sample` directory of your HyperSQL distribution.

PL # Variable

is an automatic variable just like ?. The value is set to the rowcount of the last successful query. Besides validation purposes in automation scripts, it's useful for interactive situations where result set counts are not displayed, such as when queries are run indirectly from invoked scripts with \i.

Macros

Macros are just shortcut commands that you can run in place of the full commands which they stand for. Macros stand for SQL, Special or PL commands, whereas PL variables can only be used for elements within a command. It is very easy to define, list, and use macros. Run the command "/?" to see how. If you often run a particular query, then for the effort of about 5 extra keystrokes, you can define a macro for it so that you can enter just "/q;" to run it, whether the original query is 1 line or 40 lines. (You can use any name in place of "q", and the target command can be any kind of SQL, special, or PL command).

When you run/use a macro, you can append to the macro value. *appendage* in the "/?" listing shows where you can append additional text to the original command. So, if you define

```
    sql> /= myworkers  SELECT name FROM employees
```

, you could narrow the query variously during different macro invocations, like

```
    sql> /myworkers WHERE dept = 20;
    sql> /myworkers WHERE name like 'Karen%';
```

Just like when recalling a command from history, you use ";" to execute even Special and PL macro commands.

```
    sql> /= notate  \p Work completed by
    sql> /notate Blaine;
```

If you don't type the ;, you will just recall the command to the buffer (from which you can execute or edit it, if you wish to).

To make a macro for a mult-line SQL statement, you use the "/: name" construct. First, get the target command into the command buffer. If you have already run the command, then run ":h" to see the command number and load it to the buffer like ":13". If you haven't run the command yet, then just enter the command, but end it with a blank line (and no semi-colon). You can check the buffer with ":b" to make sure it is what you want. Then just run "/: name" to define a macro with name "name".

SqlTool Functions

SqlTool functions are macros which take positional parameters. They are functions in the shell-programming sense. They do not return values in the sense of functions as distinguished from procedures or methods. As the `/?` in-program help shows, they can be defined by literal assignment or by buffer contents, and optional appendages work as one would want-- just like regular macros. They are intuitive to define and use, so one example should be all the instruction needed.

Example 1.11. Creating a SqlTool Function

```
\.
```

```
INSERT INTO t(i, vc) VALUES(*{1}, '*{2}');
SELECT * FROM t
  WHERE i = *{1}
.
/: writeread()  AND audited is null
```

This is a non-trivial example where we insert into a table with some automatically generated columns, and we want to see the entire created record before deciding whether to commit the new record. Since what we want to do will take multiple lines of SQL, and indeed 2 SQL statements, we use raw mode to write the multi-line SQL statement to the edit buffer, then use the `/: MACRONAME [appendage]` construct to define a macro with body of the previous edit buffer contents. As described elsewhere, if you want to do this in a SQL file (as opposed to interactively), you have to use raw mode as we have done here. Just by assigning a name ending with `()` we have made a function instead of a regula macro. Notice how we used positional parameters references `*{1}` and `*{2}` in the macro body. We wanted to add a little to what was in the edit buffer, so we added an appendage to the `/:` command. Note the extra space after `()` or we would have ended up with resulting body of "`... i = *{1}AND audited...`".

Example 1.12. Invoking a SqlTool Function

```
/writeread(10, ten);
```

Not much to explain. Though the second character is for a string value to insert into the a varchar column, we wrote the function so that the function body supplies the single-quotes instead of having to type them in ever time we use the function. Leading and trailing white space is trimmed from each parameter. So if you want your value to have leading or trialing space, you will have to type in the quotes at invocation time. Another limitation caused by this convenient parsing is that functions just won't work when your invocation parameters need to contain commas. Just like for regular macros, the terminating ; causes the expanded macro to execute.

PL Sample

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

Example 1.13. Simple SQL file using PL

```
    /*
       grantwrite.sql

       Run SqlTool like this:
           java -jar path/to/sqltool.jar -pUSER=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.

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).

Spaces are not allowed in elements of logical expressions. These are examples of illegal logical expressions: `* while (two words)`, `* if (*x == two words)`. You can certainly do what you want to do, however, by using variables to hold multi-word strings. You can achieve the goals for the two previous attempts with

```
    *tmpVar = two words
    * while (*tmpVar)
    ...
    * if (*x == *tmpVar)
```

It is critically important here to use `*tmpvar` instead of `*{tmpvar}` in this situation, because `*{...}` would not delay expansion and would therefore be equivalent to entering the multiple words.

SqlTool's logical expressions are purposefully minimalistic. We do not support nested operations or mixing with assignment commands. Notice that there are no `||`, `&&`, `AND`, or `OR` operations in the table below. You can not assign the value of a boolean expression directly. You can achieve that goal with an `* while` and mathematical assignments.

As stated earlier, inside of logical expressions you should normally reference PL variables without curly braces. This syntatic simplification is allowed because multi-word tokens are not allowed in logical expressions (therefore {...} is not needed to group words). For example, `"word"`, `">"`, and `"*VARNAME"` are all separate atoms.

You can indeed use the curly format like `"*{THIS}"` inside of logical expressions, but the casual user should stick to `"*THIS"`. There is a 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). Another difference between `*{VARNAME}` and `*VARNAME` is that the latter resolves to *unset* (this is very different from the empty string that `*{:VARNAME}` would resolve to).

If you do use the braces, make sure that the expansion value doesn't contain quotes or whitespace. (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. Hence-- casual users should not use braces inside of logical expressions.

Though tokens inside logical expressions are atomic, you definitely can and should do tests on strings that contain spaces. You just have to use a variable for each such string value. For example, if I want to see if the special variable `?` is equal to `one two three`, then you must do it like this:

```
    * cfString = one  two three
    * if (*cfString == ?)
```

As noted elsewhere in this guide, internal spaces are preserved as given. For assignments, trailing spaces are generally preserved. Leading spaces are preserved only for the : assignment commands.

Logical Operators

TOKEN	The token may be a literal, a `*{VARNAME}` which is expanded early, or a `*VARNAME` which is expanded late. (You usually do not want to use `*{VARNAME}` 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.
TOKEN1 >= TOKEN2	True if the TOKEN1 string is longer than TOKEN2 or is the same length but is greater or equal value according to a string sort.
TOKEN1 => TOKEN2	Ditto.
TOKEN1 <= TOKEN2	Similarly to `TOKEN1 >= TOKEN2`.
TOKEN1 =< TOKEN2	Ditto.

`*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. Another way of saying this is that `*VARNAME` in a logical expression is equivalent to *{:VARNAME} out of a logical expression.

When developing scripts, you definitely should 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.

Mathematical Assignments

Only integer math is supported, and only in mathematical assignment commands. Math assignment commands are of the format

```
<ASSIGNEE> <ASSIGNMENT_OP> <INTEGER_EXPRESSION>
```

For example,

```
SQUARE_FOOTAGE += (FOYER_FEET + 20) * 3 + 300 * BATHS
```

This works very close to Bash and Korn shell `((...))` integer math. The primary difference from those shells is that we prohibit useless non-assignment commands. Therefore, our math assignment commands always begin with the assignee variable name and an assignment operator.

Those users unfamiliar with programs that do strictly integer math should play around with it before using it for anything important. It may surprise you that real numbers like `2.9` are not automatically converted to an integer, but are simply prohibited; and that results of expressions are truncated to an integer, not rounded.

The list below is available from the program by running `* ?` (they are listed after the words "`Assignment OPs:`"). Note that though we support assignment operator `++`, we do not support `--` because that conflicts with our single-line comment delimiter `--`. The work-around is to use `-=1` instead.

Mathmatical Assignment Operators

- =
- ++ (increment by 1, no expression allowed)
- -= (subtract value of the expression)
- += (add to...)

- /= (divide by...)
- %= (divide by expression and return remainder)

To the right of the assignment operator is the integer math expression consisting of raw variable names, integers, and mathematical operators. The variables referenced, if any, must all contain integer values. In the expression only user PL variables may be used. Not Java system properties nor SqlTool system PL variables.

The list below is available from the program by running `* ?` (they are listed after the words "`Internal ops:`").

Mathmatical Expression Operators

- INTEGER_USER_VARIABLE_NAME (resolve to its value)
- () (specify precedence)
- +
- -
- *
- / (division)
- % (division remainder)
- ^ (power)

Flow Control

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

Important

Definitely run command `* ? control` to view a list of the available flow control statements, and details about how to use them.

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`).

Definitely read the section Logical Expressions .

The values of control variables for `* foreach` and `* forrows` 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).

There is now also an inline `* if` command that is very handy and concise. Try these samples on.

Example 1.14. Inline If Statement

```
    * if (*x == *NULL) \q Aborting program
    ....
```

```
    * while...
        * if (*exitCondition) * break
    ...
    * if (*notableEvent) \l SEVERE Something bad happened
```

PL Example

Below is the example SQL file `sample/pl.sql`, which shows how to use most of the basic PL features [1]. If you have a question about how to use a particular PL feature, check this file in your distribution before asking for help... and definitely read the in-program help for `* ?` carefully! Give it a run, like

```
java -jar $HSQLDB_HOME/lib/sqltool.jar mem $HSQLDB_HOME/pl.jar
```

It will suggest that you re-run it with another parameter. Insert the new parameter before "`mem`".

Example 1.15. SQL File showing use of most PL features

```
/*
    $Id: pl.sql 4564 2011-10-19 04:27:37Z unsaved $
    SQL File to illustrate the use of some basic SqlTool PL features.
    Invoke like
        java -jar .../sqltool.jar mem .../pl.sql
                                                        -- 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 yourscriptname.sql) any \q will kill their SqlTool session. */
    \p Use argument "-pMYTABLE=mytablename" for SqlTool
    * break
* end if

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

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

CREATE TABLE *{MYTABLE} (
    i int,
    s varchar(20)
);
-- PL variable ? is always set to status or fetched value of last SQL
-- statement.  It will be null/unset if the last SQL statement failed.
\p CREATE status is *{?}
\p

/* Validate our return status.
   In case of success of a CREATE TABLE, *? will be 0, and therefore a
   '* if (*?)' would be false.
   So we follow the general practice of testing *? for the error indicator
   value of null, using the reserved SqlTool system variable *NULL.
 */
* if (*? == *NULL)
    \p Our CREATE TABLE command failed.
    * break
* end if

-- Default Continue-on-error behavior is what you usually want
\c 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
* 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

/* This time instead of using the ? variable, we're assigning the SELECT value
   to a User variable, 'themax'. */
* 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};

/* No need to test for failure status (either ? or themax being unset/null),
   because we are in \c mode and would have aborted if the SELECT failed. */
* if (0 == *themax)
    \p Got 0 as the max value.
    * 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 Counting down to exit
* ((i = 3))
* while (*i > 0)
    \p *{i}...
    * ((i -= 1))  -- i++ is supported but i-- is not, because -- marks comments
* end while

\p
\p Everything worked.  Signing off.
```

Chunking

We hereby call the ability to transmit multiple SQL commands to the database in one transmission *chunking*. Normally it's best to send SQL statements to the database one-at-a-time. That way, the database can give you or your program feedback about each statement. But there are situations where it is more important to transmit multiple-statements-at-a-time than to get feedback for each statement individually.

Why?

The first general reason to chunk SQL commands is performance. For standalone databases, the most common performance bottleneck is network latency. Chunking SQL commands can dramatically reduce network traffic.

The second reason is that there are a couple SQL commands which require the terminating ";" to be sent to the database engine. For simplicity and efficiency, it's usually better for general JDBC clients like SqlTool to strip off the final delimiter. Raw commands retains everything that the user types.

The third general reason to chunk SQL commands is if your database requires you to send multiple commands in one transmission. This is usually the case with the following types of commands:

- Nested SQL commands, like the nested CREATE SCHEMA variant, and most stored procedure, function, and trigger definitions.
- Commands containing non-quoted programming language to be interpreted by the database engine. Definitions of stored procedures, function, and triggers often contain code like this.

How?

Use raw mode. Go to the Raw Mode section to see how. You can enter any text at all, exactly how you want it to be sent to the database engine. Therefore, in addition to chunking SQL commands, you can give commands for non-SQL extensions to the database. For example, you could enter JavaScript code to be used in a stored procedure.

Raw Mode

You begin raw mode by issuing the Special Command "\.". You can then enter as much text in any format you want. When you are finished, enter a line consisting of only ".;" to store the input to the edit buffer and send it to the database server for execution.

You may end the raw input with a line consisting only of "." (instead of ".;"). This will just save the input to the edit buffer so that you can do things like edit it or create a macro/function/variable for it. To execute a database command after editing, use the command ":;" when you are satisfied (use ":b" to view buffer).

You may end the raw input with a line consisting only of "." You'll notice that your prompt will be the "raw" prompt between entering "\." and terminating the raw input with ".;" or ".".

Just by running commands beginning with `BEGIN`, `DECLARE`, `CREATE function`, or `CREATE procedure`, your SqlTool session will automatically be changed to Raw mode, exactly as if you had entered "\.". That's because these commands are universally used to define stored procedures or functions, and these commands require raw mode (as explained in the previous section). You can always switch to raw mode explicitly instead of depending on the automatic switching. Raw mode always requires you to indicate where the raw input ends, regardless of raw mode was entered explicitly or automatically. Trigger definition statements do not automatically switch to raw mode, because there are many trigger definitions where raw mode is not necessary-- therefore, you must explicitly use raw mode to define triggers which contain semi-colons.

Example 1.16. Interactive Raw Mode example

```
sql> \.
Enter RAW text.  No \, :, * commands.
End with a line containing only ".;" to send to database,
or only "." to store to edit buffer for editing or saving.
------------------------------------------------------------
raw> line one;
raw> line two;
raw> line three;
raw> .
Raw chunk moved into buffer.  Run ":;" to execute the chunk.
sql> :;
Executing command from buffer:
line one;
line two;
line three;

SQL Error at 'stdin' line 13:
"line one;
line two;
line three;"
```

```
    Unexpected token: LINE
    sql>
```

The error message "Unexpected token: LINE in statement [line]" comes from the database engine, not SqlTool. All three lines were transmitted to the database engine.

Edit Buffer Commands are not available when running SqlTool non-interactively.

SQL/PSM, SQL/JRT, and PL/SQL

This section covers database-engine-embedded languages, which are often used in the definition of stored procedures, stored functions, and triggers. `SQL/PSM`, `SQL/JRT`, and `PL/SQ:` are well known examples. We prefer `SQL/PSM` and `SQL/JRT` because unlike the alternatives, they are based on open SQL specifications.

Note

PL/SQL is **not** the same as PL. PL is the procedural language of SqlFile and is independent of your back-end database. PL commands always begin with *. PL/SQL is an Oracle-specific extension processed on the server side. You can not intermix PL and any server-embedded language (except for setting a PL variable to the output of execution), because when you enter server language to SqlTool, that input is not processed by SqlFile.

Use Raw Mode to send server-language code blocks to the database engine. You do not need to enter the "\." command to enter raw mode. Just begin a new SqlTool command line with "DECLARE", "BEGIN", "CREATE FUNCTION", or "CREATE PROCEDURE", and SqlTool will automatically put you into raw mode. See the Raw Mode section for details.

The following sample SQL file resides at `sample/plsql.sql` in your HyperSQL distribution [1]. This script will only work with Oracle, only if you have permission to create the table "T1" in the default schema, and if that object does not already exist.

Example 1.17. PL/SQL Example

```
/*
 * $Id: plsql.sql 826 2009-01-17 05:04:52Z unsaved $
 *
 * This example is copied from the "Simple Programs in PL/SQL"
 * example by Yu-May Chang, Jeff Ullman, Prof. Jennifer Widom at
 * the Standord University Database Group's page
 * http://www-db.stanford.edu/~ullman/fcdb/oracle/or-plsql.html .
 * I have only removed some blank lines (in case somebody wants to
 * copy this code interactively-- because you can't use blank
 * lines inside of SQL commands in non-raw mode SqlTool when running
 * it interactively); and, at the bottom I have  replaced the
 * client-specific, non-standard command "run;" with SqlTool's
 * corresponding command ".;" and added a plain SQL SELECT command
 * to show whether the PL/SQL code worked.  - Blaine
 */

CREATE TABLE T1(
    e INTEGER,
    f INTEGER
);

DELETE FROM T1;

INSERT INTO T1 VALUES(1, 3);

INSERT INTO T1 VALUES(2, 4);
```

```
/* Above is plain SQL; below is the PL/SQL program. */
DECLARE

    a NUMBER;

    b NUMBER;

BEGIN

    SELECT e,f INTO a,b FROM T1 WHERE e>1;

    INSERT INTO T1 VALUES(b,a);

END;

.;
/** The statement on the previous line, ".;" is SqlTool specific.
 *  This command says to save the input up to this point to the
 *  edit buffer and send it to the database server for execution.
 *  I added the SELECT statement below to give imm
 */

/* This should show 3 rows, one containing values 4 and 2 (in this order)...*/
SELECT * FROM t1;
```

Note that, inside of raw mode, you can use any kind of formatting that your database engine needs or permits: Whatever you enter-- blank lines, comments, everything-- will be transmitted to the database engine.

This file resides at `testrun/sqltool/sqljrt.sql`

Example 1.18. SQL/JRT Example

```
/*
 * $Id: sqljrt.sql 5407 2014-10-13 21:16:17Z unsaved $
 *
 * Tests SQL/JRT
 */

create function dehex(VARCHAR(80), INTEGER)
    returns INTEGER
    no sql
    language java
    external name 'CLASSPATH:java.lang.Integer.valueOf'
.;

CALL dehex('12', 16);
*if (*? != 18) \q SQL/JRT function failed
```

This file resides at `testrun/sqltool/sqlpsm.sql`

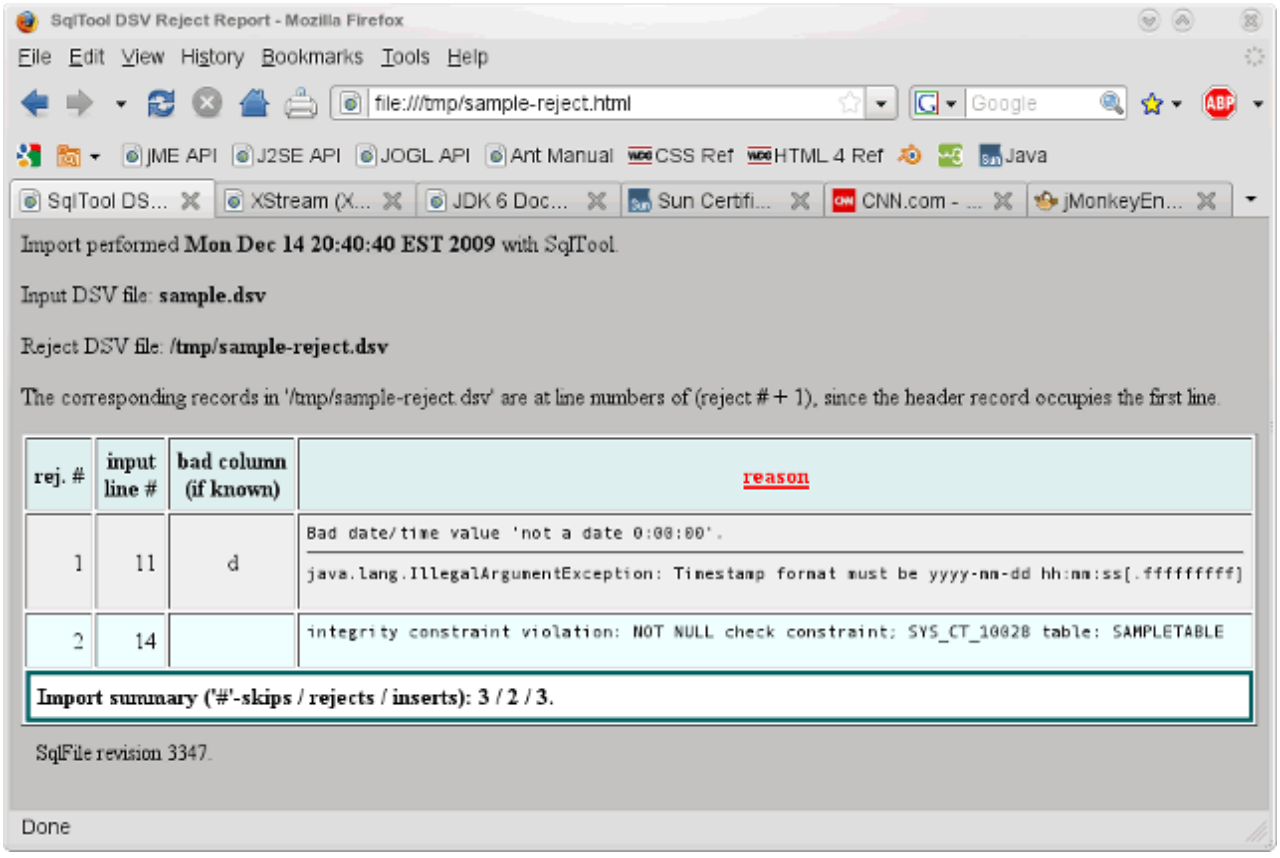
Example 1.19. SQL/PSM Example

```
/*
 * $Id: sqlpsm.sql 5407 2014-10-13 21:16:17Z unsaved $
 *
 * Tests SQL/JRT
 */

create table customers(
    id INTEGER default 0, firstname VARCHAR(50), lastname VARCHAR(50),
    entrytime TIMESTAMP);

create procedure new_customer(firstname varchar(50), lastname varchar(50))
    modifies sql data
    insert into customers values (
        default, firstname, lastname, current_timestamp)
```

```
.;

SELECT count(*) FROM customers;
*if (*? != 0) \q SQL/PSM preparation failed

CALL new_customer('blaine', 'simpson');

SELECT count(*) FROM customers;
*if (*? != 1) \q SQL/PSM procedure failed
```

Delimiter-Separated-Value Imports and Exports

SqlTool's DSV functionality encompasses what many users will recognize as CSV export, as well as portable backup or transfer of data. Those familiar with Oracle's SQL*Loader will recognize the extreme usefulness of the feature set. Besides database- and platform-independent data backups, exports can be used to deploy data sets with applications, to transfer data among multiple database instances (even drastically different database instances such as SQL Server and HyperSQL), and to properly change control data sets with a content management system such as a collaboration server or Subversion. To jump way ahead for a moment to whet your appetite, here is a sample *import reject report* which will can be generated automatically for you upon import just by setting the PL variable `*DSV_REJECT_REPORT` (to the desired destination HTML file name).



A DSV Import reject report

If you wish to, you can review the reject report before deciding whether to commit or roll back the inserts.

Note

This feature is independent of HyperSQL Text Tables. (See the Text Tables chapter of the HyperSQL User Guide [http://hsqldb.org/doc/2.0/guide/index.html] for details about them). a server-side feature

of HyperSQL. It makes no difference to SqlTool whether the source or target table of your export/import is a memory, cache, or text table. Indeed, like all features of SqlTool, it works fine with other JDBC databases. It works great, for example to migrate data from a table of one type to a table of another type, or to another schema, or to another database instance, or to another database system.

Most business type people would call this feature "CSV", but there is an important difference. Though "CSV" stands for *Comma-Separated Values*, the only thing actually distinctive about CSV is not the comma but the way that double-quotes are used for escaping purposes. As discussed in this section, with Delimiter-Separated-Value files, we purposefully choose an effective delimiter instead of the CSV method of using a delimiter which works in some cases and then use double-quoting to escape occurrence of the column-delimiter and of double-quote itself in the actual data. Just by choosing a delimiter which never needs escaping, we eliminate the whole double-quoting complication, and the data in our files always look just like the corresponding data in the database. To make this CSV / Delimiter-separated-value distinction clear, I use the suffix ".dsv" for my data files. This leads me to stipulate the abbreviation DSV for the *Delimiter Separated Value* feature of HyperSQL.

Use the `\x` command to eXport a table to a DSV file, and the `\m` command to iMport a DSV file into a pre-existing table. **Use command `\x?` or `\m?` for a listing of all related commands and options.**

The row and column delimiters may be any String (or even a regular expression for import), not just a single character. The export function is more general than just a table data exporter. Besides the trivial generalization that you may specify a view or other virtual table name in place of a table name, you can alternatively export the output of any query which produces normal text output. (This could actually even be multiple multiple-line SQL statements, as long as the last one outputs the needed data cells). A benefit to specifying even a simple query is that it allows you to export only some columns of a table, and to specify a WHERE clause to narrow down the rows to be exported (or perform any other SQL transformation, mapping, join, etc.). A specific use for this would be to exclude columns of binary data (which can be exported by other means, such as a PL loop to store binary values to files with the \bd command), or pseudo-or derived columns.

Note that the import command will not create a new table. This is because of the impossibility of guessing appropriate types and constraints based only on column names and a data sampling (which is all that a DSV-importer has access to). Therefore, if you wish to populate a new table, create the table before running the import. The import file does not need to have data for all columns of a table. The only required columns are those required by database constraints (non-null, indexes, keys, etc.) One specific reason to omit columns is if you want values of some columns to be created automatically by column DEFAULT settings, triggers, HyperSQL identity sequences, etc. Another reason would be to skip binary columns.

Due to wildly varying support and behavior of data and time types in SQL databases, SqlTool always converts date-type and time-type values being imported from DSV files using java.sql.Timestamp [http://download.oracle.com/javase/6/docs/api/java/sql/Timestamp.html]s. This usually provides more resolution than is needed, but is required for portability. Therefore, questions about acceptable date/time formats are ultimately decided by the Java's java.sql.Timestamp class [http://download.oracle.com/javase/6/docs/api/java/sql/Timestamp.html].

Simple DSV exports and imports using default settings

Even if you need to change delimiters, table names, or file names from the defaults, I suggest that you run one export and import with default settings as a practice run. A memory-only HyperSQL instance is ideal for test runs like this.

This command exports the table `icf.projects` to the file `projects.dsv` in the current directory (where you invoked SqlTool from). By default, the output file name will be the specified source table name plus the extension `.dsv`.

Example 1.20. DSV Export Example

```
SET SCHEMA icf;
\x projects
```

We could also have run `\x icf.projects` (which would have created a file named `icf.projects.dsv`) instead of changing the session schema. In this example we have chosen to make the export file name independent of the schema to facilitate importing it into a different schema.

Take a look at the output file. Notice that the first line consists of column names, not data. This line is present because it will be needed if the file is to used for a DSV import. Notice the following characteristics about the export data. The column delimiter is the pipe character "|". The record delimiter is the default line delimiter character(s) for your operating system. The string used to represent database `NULL`s is `[null]`. See the next section for how to change these from their default values.

Warning

You can not DSV import Array values where any Array elements contain commas, for example an Array of VARCHARs which contain one or more commas. There is no such limitation on DSV exports, which you can use for purposes other than SqlTool importing, or you could use a script to change the commas to some other character.

This command imports the data from the file `projects.dsv` in the current directory (where you invoked SqlTool from) into the table `newschema.projects`. By default, the output table name will be the input filename after removing optional leading directory and trailing final extension.

Example 1.21. DSV Import Example

```
    SET SCHEMA newschema;
    \m projects.dsv
```

If the DSV file was named with the target schema, you would have skipped the `SET SCHEMA` command, like `\m newschema.projects.dsv`. In order to allow for more flexibility, the default input input delimiters are not exactly the same as the output delimiters. The input delimiters are regular expressions. The input column delimiter happens to be the regular expression corresponding exactly to "|"; but the input record delimiter matches UNIX, Windows, Mac, and HTTP line breaks.

Specifying queries and options

For a hands on example of a DSM import which generates an import report and uses some other options, change to directory `HSQLDB/sample` and play with the working script `dsv-sample.sql` [1]. You can execute it like

```
    java -jar ../lib/sqltool.jar mem dsv-sample.sql
```

(assuming that you are using the supplied `sqltool.rc` file or have have urlid `mem` set up).

The header line in the DSV file is required at this time. (If there is user demand, it can be made optional for exporting, but it will remain required for importing).

Your export will fail if the output column or record delimiter, or the null representation value occurs in the data being exported. You change these values by setting the PL variables `*DSV_COL_DELIM`, `*DSV_ROW_DELIM`, `*DSV_NULL_REP`. Notice that the asterisk is part of the variable names, to indicate that these variables are used by SqlTool internally. Regular expressions have their own mechanism for including special characters. `*DSV_NULL_REP` effects normal displaying of VARCHAR output to screen or stdout, not just importing and exporting-- so you should reset the value if you want to revert to normal display behavior. When specifying output delimiters, you can use the escape sequences \n, \r, \f, \t, \\, and decimal, octal or hex specifications like \20, \020, \0x20. For example, to change the column delimiter to the tab character, you would give the command

```
    * *DSV_COL_DELIM = \t
```

The input (\m) delimiter values, `*DSV_COL_SPLITTER` and `*DSV_ROW_SPLITTER`, are set using normal Perl/ Java regexp syntax. There are escapes for specifying special characters, and anything else you would need. Input vs.

output row and column delimiters are easily distinguished by containing "SPLITTER" for splitting input (\m) files; or "DELIM" for the delimiters that we will write (\x) among the data.

*DSV...DELIM vs *DSV...SPLITTER settings

Both the ...DELIM and the ...SPLITTER settings are for delimiting cells of data, but whereas our DELIM values are literal things that SqlTool will write right into a DSV file, SPLITTER values are patterns for detecting the literal delimiters in existing DSV files.

The settings named like `*DSV...SPLITTER` are input delimiters specified as regular expressions following the rules in the API spec for java.util.regex.Pattern [http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html]. The settings named like `*DSV...SPLITTER` are output delimiters specified as constant strings which can contain escape sequences to represent special characters (as documented in this section).

For imports, you must always specify the source DSV file path. If you want to *export* to a different file than one in the current directory named according to the source table, set the PL variable `*DSV_TARGET_FILE`, like

```
    * *DSV_TARGET_FILE = /tmp/dtbl.dsv
```

For exports, you must always specify the source table name or query. If you want to *import* to a table other than that derived from the input DSV file name, set the PL variable `*DSV_TARGET_TABLE`. The table name may contain a schema name prefix.

You don't need to import all of the columns in a data file. To designate the fields to be skipped, iether set the PL PL variable `*DSV_SKIP_COLUMNS`, or replace the column names in the header line to "-" (hyphen). The value of `*DSV_SKIP_COLUMNS` is case-insensitive, and multiple column names are separated with white space and/or commas.

You can specify a query instead of a tablename with the \x command in order to filter or transform data from a table or view, or to export the output of a join, etc. You must set the PL variable `*DSV_TARGET_FILE`, as explained above (since there is no table name from which to automatically map a file name).

Example 1.22. DSV Export of an Arbitrary Query

```
    * *DSV_TARGET_FILE = outfile.txt
    \x SELECT entrydate, 2 * aval "Double aval", modtime FROM bs.dtbl
```

Note that I specified the column label alias "Double aval" so that the label for that column in the DSV file header will not be blank. You can type a query line as long long as you want to, but if you want to use a specified query that spans multiple lines, then you must use the command variant `\x :` to use the query in the previous edit buffer. (To populate the edit buffer with your multi-line SQL query, you must execute the command before... usually undesirable, or end the SQL with a blank line instead of a `;`... only works interactively, or use Raw Mode).

By default, imports will abort as soon as a error is encountered during parsing the file or inserting data. If you invoke SqlTool with a SQL script on the command line, the failure will cause SqlTool to roll back and exit. If run interactively, you can decide whether to commit or roll back the rows that inserted before the failure. You can modify this behavior with the \a and \c settings.

If you set either a reject dsv file or a reject report file, then failures during imports will be reported but will not cause the import to abort. When run in this way, SqlTool will give you a report at the end about how many records were skipped, rejected, and successfully inserted. The reject dsv file is just a dsv file with exact copies of the dsv records that failed to insert. The reject report file is a HTML report which lists, for every rejected record, why that record was rejected. `\m?` will show you that the required PL variables for this functionality are `*DSV_REJECT_FILE` and `*DSV_REJECT_REPORT`. In both cases, you set the variable value to the path of the file which SqlTool will create.

Reject reports use the same templating system as SqlTool HTML reports. Therefore you can set SqlTool system PL variables `*TOP_HTMLFRAG_FILE` or `*BOTTOM_HTMLFRAG_FILE` to use your own opening and closing HTML and to completely replace the styling. If you use the default templates you can set user PL variable `REPORT_TITLE` for the obvious reason, and you can place a file named `overrides.css` into the same directory as your generated report, for the obvious purpose. You can use PL variable references in your own fragment files (remember to use the `${:VARNAME}` construct to prevent errors for variables that are not set). You can also use automatically set variables like `*TIMESTAMP` and `*REVISION`

To allow for user-friendly entry of headers, we require that tables for DSV import/exports use standard column names. I.e., no column names that would require quoting in interactive SQL statements. The DSV import and export parsers are very smart and user-friendly. The data types of columns are checked so that the parser can make safe assumptions about white space and blank entries in the data. If a column is a JDBC Boolean type, for example, then we know that a field value of " True " obviously means "True", and that a field value of "" obviously means null. Since we require vanilla style column names, we allow white space anywhere in the header column. We allow blank lines anywhere (where "lines" are delimited by `*DSV_ROW_DELIM`). By default, commented lines are ignored, but this can be disabled (by setting `DSV_SKIP_PREFIX` to the empty string) or you can change the delimiter character from # to whatever you want (by setting `DSV_SKIP_PREFIX` to that value).

Use In-Program Help for Importing and Exporting

Run the command "\x?" or "\m?" to see the several system PL variables which you can set to adjust reject file behavior, commenting behavior, and other DSV features. The in-program help is the definitive reference for available options, not this manual.

You can also define some settings right in the DSV file, and you can even specify multiple header lines in a single DSV file. I use this last feature to import data from one data set into multiple tables that are joined. Since I don't have any more time to dedicate to explaining all of these features, I'll give you some examples from working DSV files and let you take it from there.

Example 1.23. Sample DSV headerswitch settings

```
    # RCS keyword was here.

    headerswitch{
    itemdef:name|-|-|hardness|breakdc|-
    simpleitemdef:itemdef_name|maxvalue|weight|-|-|maxhp
    }
```

I'll just note that the prefixes for the header rows must be of format target-table-name + :. You can use * for target-table-name here, for the obvious purpose.

Example 1.24. DSV targettable setting

```
    targettable=t
```

This last example is from the SqlTool unit test file `dsv-trimming.dsv`. These special commands must be at the top of the file (before any normal data or header lines).

There is also the `*DSV_CONST_COLS` setting, which you can use to automatically write static, constant values to the specified columns of all inserted rows.

CSV Imports and Exports

The only difference between CSV and DSV is that CSVs allow presence of the column delimiter in the CSV file, and require the use of double-quotes to escape occurrences of both that column delimiter and of double-quotes in the real

data cells. To enable this double-quote escaping, just use commands `\xq` and `\mq` instead of `\x` and `\m`. Since CSV *is* this double-quote escaping, SqlTool's `\xq` and `\mq` commands initiate CSV exports and imports. Conflicting with the name, CSV files do not need to use comma as the column delimiter, and the tab character is a common alternative. Since CSV importing and exporting is implemented as a sub-case of DSV import and exporting, everything in the Delimiter-Separated-Value Imports and Exports section applies, and CSV users should definitely read that section.

I should also mention the trivial difference between `\xq` and `\x` that if you do not specify `*DSV_TARGET_FILE`, the default filename suffix will be "`.csv`" instead of "`.dsv`".

Always use command \mq? or \xq? to list all available import and export options.

Settings Often of Interest to CSV User

*DSV_COL_DELIM — Set to what column delimiter to write to the CSV file. Values of "`,`" and "`\t`" (without the quotes) are most common with CSVs. This is what SqlTool will use to *separate* the values in a single output CSV file line.

*DSV_COL_SPLITTER — Set to the column-delimiter character in the CSV file to be read. This is what SqlTool will use to *split* each line into multiple cell values. Values of "`,`" and "`\t`" (without the quotes) are most common with CSVs.

*NULL_REP_TOKEN — This effects only data coming from or destined to columns with a string data type, because nulls can easily be distinguished from non-nulls for other data types. By default, SqlTool will distinguish between nulls and empty strings for string columns. Many CSV-support applications can't handle importing or exporting nulls. In you are interfacing to such an app, set Java system property '`sqltool.REMOVE_EMPTY_VARS`' to `false` and set `*NULL_REP_TOKEN` to the empty string like "`* *NULL_RP_TOKEN =`". This will cause both nulls and empty strings to write empty strings to the export CSV file; and will cause empty strings in the import CSV file to create nulls. (The Java system property setting will become unnecessary with the next minor relase of SqlTool because that is going to be SqlTool's default behavior). `*NULL_REP_TOKEN` also effects how nulls in VARCHAR columns are represented in regular query output (non-exports), so after your exporting/importing you will often want to reset it with "`* - *NULL_REP_TOKEN`" unless you will be exiting SqlTool immediately.

*ALL_QUOTED — Every cell value will be quoted upon \xq, instead of just those values containing a column delimiter character or double-quote that needs escaping. `*ALL_QUOTED` does not effect the `*NULL_REP_TOKEN`. If you want the null-rep token to be double-quoted took, then you must set the `*NULL_REP_TOKEN` value itself to be double-quoted.

*DSV_REJECT_REPORT — Set this to the path of a HTML file that will be generated if any bad input records are encountered upon \mq. Instead of aborting, SqlTool will continue and import every record that it is able to. You can view the summary counts (always displayed) and/or the reject report before deciding whether to commit or rollback the new database records.

This sample shows everything you need to know to get going with CSV. `csv-sample.sql` [1].

Example 1.25. Sample CSV export + import script

```
/*
 * $Id: csv-sample.sql 4812 2011-11-20 21:31:49Z unsaved $
 *
 * Create a table, CVSV-export the data, import it back.
 */

* *DSV_COL_DELIM = ,
```

```
* *DSV_COL_SPLITTER = ,
-- Following causes a reject report to be written if there are any bad records
-- during the import.  To test it, enable the "FORCE AN ERROR" block below.
* *DSV_REJECT_REPORT = import.html

-- 1. SETTINGS
-- For applications like MS Excel, which can't import or export nulls, we have
-- to dummy down our database empty strings to export and import as if they
-- were nulls.
* *NULL_REP_TOKEN =

-- Enable following line to quote every cell value
-- * *ALL_QUOTED = true


-- 2. SET UP TEST DATA
CREATE TABLE t (i INT, v VARCHAR(25), d DATE);
INSERT INTO t(i, v, d) VALUES (1, 'one two three', null);
INSERT INTO t(i, v, d) VALUES (2, null, '2007-06-24');
INSERT INTO t(i, v, d) VALUES (3, 'one,two,,three', '2007-06-24');
INSERT INTO t(i, v, d) VALUES (4, '"one"two""three', '2007-06-24');
INSERT INTO t(i, v, d) VALUES (5, '"one,two"three,', '2007-06-24');
INSERT INTO t(i, v, d) VALUES (6, '', '2007-06-24');
commit;

-- 3. CSV EXPORT
/* Export */
\xq t
/*  FORCE AN ERROR.  Enable the following 3 lines to force a bad CSV record.
\o t.csv
\p barf
\o
*/

-- 4. BACK UP AND ZERO SOURCE TABLE
CREATE TABLE orig AS (SELECT * FROM t) WITH DATA;
DELETE FROM t;
commit;

-- 5. CSV IMPORT
\mq t.csv
commit;

-- 6. MANUALLY EXAMINE DIFFERENCES BETWEEN SOURCE AND IMPORTED DATA.
-- See <HSQLDB_ROOT>/testrun/sqltool/csv-roundtrip.sql to see a way to make
-- this same comparison programmatically.
* - *NULL_REP_TOKEN
\p
\p ORIGINAL:
SELECT * FROM orig;
\p
\p IMPORTED:
SELECT * FROM t;
\p
\p The empty string in the source table will have been translated to null in
\p the imported data.
\p You can see that the generated CSV file represents both nulls and
\p empty strings as nothing, hence the convergence.
```

Unit Testing SqlTool

The SqlTool unit tests reside at `testrun/sqltool` in your HyperSQL distribution or source code repository. Just run `runtests` in that directory to execute all of the tests (except for non-Windows, non-UNIX, non-MacOS users, who must invoke `../../build/gradlew` directly). Read the file `README.txt` to find out all about file naming conventions so that you can write your own SQL test script files.

The system requirements to run the tests is now just a Java 6 JRE. The real test runner is implemented in the Groovy script `runtests.groovy`. By just typing `runtests`, Windows and Linux (incl. MacOS) users will invoke their OS-specific scripts. All users can invoke Gradle manually instead if they wish to, using either `../../build/gradlew` or a local Gradle installation. If you have Groovy installed, you can cut out all of the wrappers and invoke the Groovy script directly, like `groovy runtests.groovy` (or change the interpreter line within the script file to point to your own `groovy` path).

Chapter 2. Hsqldb Test Utility

The `org.hsqldb.test` package contains a number of tests for various functions of the database engine. Among these, the `TestUtil` class performs the tests that are based on scripts. To run the tests, you should compile the `hsqldbtest.jar` target with Ant and JUnit.

The `TestUtil` class should be run in the /testrun/hsqldb directory of the distributed files. It then runs the set of TestSelf*.txt files in the directory. To start the application in Windows, change to the directory and type:

```
java org.hsqldb.test.TestUtil
```

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.

The class `org.hsqldb.test.TestScriptRunner` is a more general program which you can use to test any script files which you specify (with scripts of the same exact format as described below). For example,

```
java org.hsqldb.test.TestScriptRunner --urlid=mem script1.tsql script2.sql
```

You must have the HSQLDB classes, including the util and test classes, in your `CLASSPATH`. The urlid must be set up in an RC file as explained in the RC File Authentication Setup section. Use the `rcfile=` argument to specify an RC file other than the default of `testscriptrunner.rc` in the current directory. To see all invocation possibilities, just run TestScriptRunner with no arguments at all. TestScriptRunner can run tests sequentially (the default) or in simultaneous asynchronous threads.

- Comment lines must start with -- and are ignored
- Lines starting with spaces are the continuation of the previous line (for long SQL statements)
- SQL statements with no prefix are simply executed.
- *The remaining items in this list exemplify use of the available command line-prefixes.*
- The /*s*/ option stands for silent. It is used for executing queries regardless of results. Used for preparation of tests, not for actual tests.

  ```
  /*s*/ Any SQL statement - errors are ignored
  ```

- The /*c<rows>*/ option is for SELECT queries and asserts the number of rows in the result matches the given count.

  ```
  /*c<rows>*/ SQL statement returning count of <rows>
  ```

- The /*u*/ option is for queries that return an update count, such as DELETE and UPDATE. It asserts the update count matches.

  ```
  /*u<count>*/ SQL statement returning an update count equal to <count>
  ```

- The /*e*/ option asserts that the given query results is an error. It is mainly used for testing the error detection capabilities of the engine. The SQL State of the expected error can be defined, for example /*e42578*/, to verify the returned error. This option can be used with syntactically valid queries to assert a certain state in the database. For example a CREATE TABLE can be used to assert the table of the same name already exists.

  ```
  /*e*/ SQL statement that should produce an error when executing
  ```

- The /*r....*/ option asserts the SELECT query returns a single row containing the given set of field values.

```
/*r<string1>,<string2>*/ SQL statement returning a single row ResultSet equal to the specified
 value
```

- The extended /*r...*/ option asserts the SELECT query returns the given rows containing the given set of field values.

```
/*r
    <string1>,<string2>
    <string1>,<string2>
    <string1>,<string2>
*/ SQL statement returning a multiple row ResultSet equal to the specified values
```

  (note that the result set lines are indented).

- The /*d*/ directive just displays the supplied text.

```
/*d*/ Some message
```

- The /*w MILLIS*/ directive causes the test to Wait for a specified number of millisedonds.

```
/*w 1000*/ Optional message
```

- The /*w ENFORCE_SEQUENCE WAITER_NAME*/ directive causes the test to Wait for the specified *Waiter*. A waiter is just name that is used to associate a /*w*/ directive to its corresponding /*p*/ directive. The ENFORCE_SEQUENCE argument must be set to `true` or `false` to specify whether to fail unless the /*p*/ command runs after the /*w*/ command is waiting.

```
/*w true script4*/ Optional message
```

- The /*p ENFORCE_SEQUENCE WAITER_NAME*/ directive is the peer directive to /*w*/, which causes a waiting thread to Proceed.

```
/*p true script4*/ Optional message
```

- All the options are lowercase letters. During development, an uppercase can be used for a given test to exclude a test from the test run. The utility will just report the test blocks that have been excluded without running them. Once the code has been developed, the option can be turned into lowercase to perform the actual test.

See the TestSelf*.txt files in the /testrun/hsqldb/ directory for actual examples.

The String `${timestamp}` may be used in script messages (like in /*d*/, /*w*/, /*p*/). It expands to the current time, down to the second. For example,

```
/*d*/ It is now ${timestamp}
```

Chapter 3. Database Manager

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

$Revision: 5063 $
2015-06-29 22:28:08-0400

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 SWING version has more refinements than the AWT version.

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 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 labelled 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.

Most SWING menu items have context-sensitive tool tip help text which will appear if you hold the mouse cursor still over the desired menu item. (Assuming that you don't turn Tooltips off under the Help menu.

The database object tree in the SWING version allows you to right click on the name of a table or column and choose from common SQL statements for the object, for example SELECT * FROM thistable ... If you click on one of the given choices, the sample statement is copied to the command window, where you can modify and complete it.

The DatabaseManagers do work with HSQLDB servers serving TLS-encrypted JDBC data. See the TLS section of the Listeners chapter of the HyperSQL User Guide [distro_baseurl_DEFAULTVAL/guide/index.html]

Tip

If you are using DatabaseManagerSwing with Oracle, you will want to make sure that Show row counts and Show row counts are both off *before connecting to the database*. You may also want to turn off Auto tree-update, as described in the next section.

Auto tree-update

By default, the object tree in the left panel is refreshed when you execute DDL which may update those objects. If you are on a slow network or performance-challenged PC, use the view / Auto-refresh tree menu item to turn it off. You will then need to use the viewRefresh tree menu item every time that you want to refresh the tree.

Note

Auto-refresh tree does not automatically show all updates to database objects, it only refreshes when you submit DDL which may update database objects. (This behavior is a compromise between utility and performance).

Automatic Connection

You can use command-line switches to supply connection information. If you use these switch(es), then the connection dialog window will be skipped and a JDBC connection will be established immediately. Assuming that the `hsqldb.jar` (or an alternative jar) are in your `CLASSPATH`, this command will list the available command-line options.

```
    java org.hsqldb.util.DatabaseManagerSwing --help
```

It's convenient to skip the connection dialog window if you always work with the same database account.

Warning

Use of the --password switch is not secure. Everything typed on command-lines is generally available to other users on the computer. The problem is compounded if you use a network connection to obtain your command line. The RC File section explains how you can set up automatic connections without supplying a password on the command line.

RC File

You can skip the connection dialog window securely by putting the connection information into an RC file and then using the `--urlid` switch to DatabaseManager or DatabaseManagerSwing. This strategy is great for adding launch menu items and/or launch icons to your desktop. You can set up one icon for each of the database accounts which you regularly use.

The default location for the RC file is `dbmanager.rc` in your home directory. The RC File Authentication Setup section explains how to put the connection information into this text file. If you also run SqlTool , then you can share the RC file with SqlTool by using a sym-link (if your operating system supports sym links), or by using the `--rcfile` switch for either SqlTool or DatabaseManagerSwing.

Warning

Use your operating system facilities to prevent others from reading your RC file, since it contains passwords.

To set up launch items/icons, first experiment on your command line to find exactly what command works. For example,

```
java -cp /path/to/hsqldb.jar org.hsqldb.util.DatabaseManagerSwing --urlid mem
```

Then, use your window manager to add an item that runs this command.

Using the current DatabaseManagers with an older HSQLDB distribution.

This procedure will allow users of a legacy version of HSQLDB to use all of the new features of the DatabaseManagers. You will also get the new version of the SqlTool! This procedure works for distros going back to 1.7.3.3 at least, probably much farther.

These instructions assume that you are capable of running an Ant build. See the Building Appendix of the HyperSQL User Guide [distro_baseurl_DEFAULTVAL/guide/index.html].

1. Download and extract a current HSQLDB distribution. If you don't want to use the source code, documentation, etc., you can use a temporary directory and remove it afterwards.
2. Cd to the build directory under the root directory where you extracted the distribution to.
3. Run `ant hsqldbutil`.
4. If you're going to wipe out the build directory, copy `hsqldbutil.jar` to a safe location first.
5. For now on, whenever you are going to run DatabaseManager*, make sure that you have this `hsqldbutil.jar` as the first item in your `CLASSPATH`.

Here's a UNIX example where somebody wants to use the new DatabaseManagerSwing with their older HSQLDB database, as well as with Postgresql and a local application.

```
CLASSPATH=/path/to/hsqldbutil.jar:/home/bob/myapp/classes:/usr/local/lib/pg.jdbc3.jar
export CLASSPATH
java org.hsqldb.util.DatabaseManagerSwing --urlid urlid
```

DatabaseManagerSwing as an Applet

DatabaseManagerSwing is also an applet. You can use it in HTML, JSPs, etc. Be aware that in Applet mode, actions to load or save local files will be disabled, and attempts to access any server other than the HTML-serving-host will fail.

Since the Applet can not store or load locally saved preferences, the only way to have persistent preference settings is by using Applet parameters.

DatabaseManagerSwing Applet Parameters

jdbcUrl	URL of a data source to auto-connect to. String value.
jdbcDriver	URL of a data source to auto-connect to. String value. Defaults to `org.hsqldb.driver.JDBCDriver`.
jdbcUser	User name for data source to auto-connect to. String value.
jdbcPassword	Password for data source to auto-connect to. String value. Defaults to zero-length string.
schemaFilter	Display only object from this schema in the object navigator. String value.
laf	Look-and-feel. String value.
loadSampleData	Auto-load sample data. Boolean value. Defaults to false.
autoRefresh	Auto-refresh the object navigator when DDL modifications detected in user SQL commands. Boolean value. Defaults to true.

showRowCounts	Show number of rows in each table in the object navigator. Boolean value. Defaults to false.
showSysTables	Show system tables in the object navigator. Boolean value. Defaults to false.
showSchemas	Show object names like schema.name in object navigator. Boolean value. Defaults to true.
resultGrid	Show query results in Gui grid (as opposed to in plain text). Boolean value. Defaults to true.
showToolTips	Show help hover-text. Boolean value. Defaults to true.

Chapter 4. Transfer Tool

Fred Toussi, The HSQL Development Group

$Revision: 5063 $
2015-06-29 22:28:08-0400

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.

From version 1.8.0 Transfer Tool is no longer part of the `hsqldb.jar`. You can build the `hsqldbutil.jar` using the Ant command of the same name, to build a jar that includes Transfer Tool and the Database Manager.

When collecting meta-data, Transfer Tool performs SELECT * FROM <table> queries on all the tables in the source database. This may take a long time with some database engines. When the source database is HSQLDB, this means memory should be available for the result sets returned from the queries. Therefore, the memory allocation of the java process in which Transfer Tool is executed may have to be high.

The current version of Transfer is far from ideal, as it has not been actively developed for several years. The program also lacks the ability to create UNIQUE constraints and creates UNIQUE indexes instead. However, some bugs have been fixed in the latest version and the program can be used with most of the supported databases. The best way to use the program is the DUMP and RESTORE modes, which allow you to manually change the SQL statements in the dump file before restoring to a database. A useful idea is to dump and restore the database definition separately from the database data.

Appendix A. SqlTool System PL Variables

As of SqlFile revision 5448

SqlTool System PL variables are the mechanism used to configure SqlTool behavior. You can list all *set* PL variables by running the SqlTool command `* listvalues`. If a SqlTool System variable is not shown, then it is *unset* (which is equivalent to *non-null*). But if a system variable is not set, that doesn't mean that the setting behavior will be *unset*, but rather that the *default behavior* will apply. For example, if you `* listvalues` and the variable `*DSV_COL_DELIM` is not listed, that doesn't mean that there will be no DSV column delimiter, but that the default DSV column delimiter will be used. The in-program help can be used to determine what the default behavior is. (In the case of `*DSV_COL_DELIM`, you can see the default behavior by running `\x?`.

Besides System PL variables, there are also user PL variables which have names beginning with a letter, and the special variables `?` and `NULL`. `NULL` is completely equivalent to `*NULL`, which is explained below. See the SqlTool chapter about `?`.

*ALL_QUOTED	Boolean. Whether to quote all values (excluding null values) in a \xq export. No effect on any command other than \xq.
*BOTTOM_HTMLFRAG_FILE	File path to HTML fragment file to close the HTML report or DSV import reject file.
*DSV_COL_DELIM	Literal string (which may contain character escapes). DSV or CSV output column delimiter literal. Run \x? to see default value and details.
*DSV_COL_SPLITTER	Regular expression. DSV or CSV input column delimiter regular expression. Run \x? to see default value and details.
*DSV_CONST_COLS	A list of `column_name = column_value|...` settings. Specifies constant import values. Run \x? to see default value and details.
*DSV_RECORDS_PER_COMMIT	Integer. How often to commit upon DSV/CSV imports. Run \x? to see default value and details.
*DSV_REJECT_FILE	File path. Path to DSV file of rejects rejected upon CSV/DSV imports. Run \x? to see default value and details.
*DSV_REJECT_REPORT	File path. Path to HTML report about CSV/DSV import failures. Run \x? to see default value and details.
*DSV_ROW_DELIM	Literal string (which may contain character escapes). DSV or CSV output row delimiter literal. Run \x? to see default value and details.
*DSV_ROW_SPLITTER	Regular expression. DSV or CSV input row delimiter regular expression. Run \x? to see default value and details.
*DSV_SKIP_COLS	A list of column names to skip, like `column1|column2` Specifies columns to omit from CSV or DSV importing or exporting. Run \x? to see default value and details.
*DSV_SKIP_PREFIX	Literal string (which may contain character escapes). Specifies comment delimiter character or string in DSV or CSV files. Run \x? to see default value and details.
*DSV_TRIM_ALL	Boolean. Trim leading and trailing white space from every cell in CSV or DSV file upon import. Run \x? to see default value and details.

*DSV_TARGET_FILE	File path. File where to export CSV or DSV to. Run \x? to see default value and details.
*DSV_TARGET_TABLE	Table name. Table where to import CSV or DSV to. Run \x? to see default value and details.
*IGNORE_BANG_STATUS	Boolean. If true, then if an external command executed by `\!` returns error (non-zero) status, SqlTool will not report or try to act on the error. (This will have no effect on what the external program may do).
*NULL	Null (i.e. always unset).
*NULL_REP_HTML	Literal string (which may contain character escapes). Same as `*NULL_REP_TOKEN`, but only applies to HTML reports.
*NULL_REP_TOKEN	Literal string (which may contain character escapes). String value to represent SQL nulls from VARCHAR columns and null (unset) PL variable values. Applies to what displays on screen and what gets written into export files.
*REVISION	Read only. Literal string.
*ROW	Read only. Literal string. Set only inside of `* forrow` loop bodies. If there is only a single column fetched, then this is equal to that cell of the current row, unless that value is null, in which case `*ROW` will be the `*NULL_REP_TOKEN` value.
*START_TIME	Read only. Literal string. Automatically set to a localized string presenting the date and time.
*TIMESTAMP	Read only. Literal string. Only usable if `*TIMESTAMP_FORMAT` has been set. Displays the date and/or time at which this variable is dereferenced.
*TIMESTAMP_FORMAT	Formatting string, as described below. Setting this variable enables the `*TIMESTAMP` read-only variable to be used. Set to a date and/or time format like `yyyy-MM-dd'T'HH:mm:ss.SSSZ`, as described at http://download.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html
*TOP_HTMLFRAG_FILE	File path. File path to HTML fragment file to open the HTML report or DSV import reject file.

Appendix B. HyperSQL File Links

HyperSQL Files referred to in this Guide

HyperSQL files referred to in the text may be retrieved from the canonical HyperSQL documentation site, http://hsqldb.org/doc/2.0, or from the same location you are reading this page from.

Note

If you are reading this document with a standalone PDF reader, only the http://hsqldb.org/doc/2.0/... links will function.

Pairs of local + http://hsqldb.org/doc/2.0 links for referenced files.

- Local: ../verbatim/sample/sqltool.rc

  http://hsqldb.org/doc/2.0/verbatim/sample/sqltool.rc

- Local: ../verbatim/sample/sampledata.sql

  http://hsqldb.org/doc/2.0/verbatim/sample/sampledata.sql

- Local: ../verbatim/sample/sample.sql

  http://hsqldb.org/doc/2.0/verbatim/sample/sample.sql

- Local: ../verbatim/sample/html-report.sql

  http://hsqldb.org/doc/2.0/verbatim/sample/html-report.sql

- Local: ../verbatim/sample/pl.sql

  http://hsqldb.org/doc/2.0/verbatim/sample/pl.sql

- Local: ../verbatim/sample/plsql.sql

  http://hsqldb.org/doc/2.0/verbatim/sample/plsql.sql

- Local: ../verbatim/sample/dsv-sample.sql

  http://hsqldb.org/doc/2.0/verbatim/sample/dsv-sample.sql

- Local: ../verbatim/sample/csv-sample.sql

  http://hsqldb.org/doc/2.0/verbatim/sample/csv-sample.sql

- Local: ../verbatim/testrun/sqltool/sqljrt.sql

  http://hsqldb.org/doc/2.0/verbatim/testrun/sqltool/sqljrt.sql

- Local: ../verbatim/testrun/sqltool/sqlpsm.sql

  http://hsqldb.org/doc/2.0/verbatim/testrun/sqltool/sqlpsm.sql

- Local: ../verbatim/src/org/hsqldb/sample/SqlFileEmbedder.java

  http://hsqldb.org/doc/2.0/verbatim/src/org/hsqldb/sample/SqlFileEmbedder.java

- Local: ../apidocs/org/hsqldb/cmdline/SqlFile.html

  http://hsqldb.org/doc/2.0/apidocs/org/hsqldb/jcmdline/SqlFile.html