Ch. 8. Manuel de SqlTool

De Apache OpenOffice Wiki
< FR | Documentation | HSQLDB Guide

Sommaire

L'Outil SQL

Buts, Changements de comportements

Ce document explique comment utiliser sqlTool. Son but principal est de lire vos fichiers textes SQL ou stdin et exécuter les commandes SQL à cet égard contre une base de donnée JDBC. SqlTool a également un grand nombre de caractéristiques pour faciliter à la fois l'usage interactif et l'automation. Les paragraphes suivants expliquent, d'une manière générale, pourquoi SqlTool est meilleur que les quelques outils existants pour le travail interactif de SQL en mode texte et pour les tâches automatisées. SqlTool partage deux bénéfices importants avec les autres outils JDBC en pur Java. Le premier est de fournir aux utilisateurs une interface et une syntaxe conséquentes pour interagir avec une immense variété de bases de données, toutes les bases de données supportant JDBC. Le second est que l'outil lui-même fonctionne sur toutes les plateformes Java. Au lieu d'utiliser `isql` pour Sybase, `psql` pour Postgresql, `sql*plus` pour Oracle, etc ... vous pouvez utiliser SqlTool pour chacune d'entre elles. Aussi loin que portent mes connaissances, SqlTool est le seul à être : prêt pour la production, pur Java, en ligne de commande et client générique JDBC. Différentes bases de données se présentent avec une interface en ligne de commande avec des capacités JDBC limitées (habituellement destinées pour un usage dans leur base de donnée spécifique).

SqlTool n'est résolument pas une interface graphique de programmation [Gui tool] comme Toad ou DatabaseManager. Il y a de nombreux cas où une interface graphique de programmation pour un outil SQL pourrait être préférable. Là où une automation est requise, vous avez réellement besoin d'un

[text client] pour tester proprement les objets et le prototype du programme. Un outil en ligne de commande est réellement supérieur pour exécuter les scripts SQL, l'automation, la recherche directe de fichiers et les accès distants. Pour clarifier cela, si vous avez à faire votre travail SQL « client » à partir d'un serveur par une connexion VPN vous apprécierez rapidement la différence de vitesse entre la transmission de données en interface texte et la transmission de données en interface graphique, même si vous utilisez VNC ou Remote Console. Vous l'apprécierez également lorsque vous ferez des tâches répétitives ou très structurées où variables et caractéristiques du langage devront être utilisées. Les partisans d'une interface graphique peuvent être en désaccord avec moi. Mais l'écriture de scripts (ou autres) est plus efficiente qu'un copié-collé répétitif avec un interface graphique. SqlTool démarre très rapidement et occupe une portion congrue de mémoire vive en comparaison avec une interface graphique complexe telle Toad.

SqlTool est supérieur pour une utilisation interactive, car depuis de nombreuses années beaucoup des caractéristiques élaborées se sont révélée efficaces pour une utilisation quotidienne. Trois commandes d'aide (\?, :?, et *?) listent toutes les commandes disponibles. SqlTool ne prend pas en charge flèche-haut ou d'autres séquences d'échappements "maison" [SqlTool doesn't support up-arrow or other OOBescapes] (à cause des limitations de base des entrées-sorties Java). Mais il fait plus que compenser cette limitation avec des alias, des variables utilisateur, l'historique et le rappel des lignes de commandes et l'édition en ligne de commande avec les expressions régulières en Perl/Java étendu. La commande \d livre les informations sur les métadonnées JDBC de façon aussi cohérente que possible (dans certains cas, des bases de données spécifiques sont utilisées pour obtenir les données sous-jacentes, même si la base de données ne fournit pas de métadonnées selon les spécifications JDBC). Contrairement aux caractéristiques d'un langage côté serveur, la même fonctionnalité fonctionne pour tout serveur de base de données. Les détails des accès aux bases de données peuvent être effectués en ligne de commande, mais les utilisateurs au quotidien voudront centraliser les détails de la connexion JDBC en un seul fichier protégé RC. Vous pouvez mettre les détails de la connexion dans votre fichier RC pour évaluer les bases de données cibles (nom d'utilisateur, mot de passe, URL, et autres paramètres facultatifs), puis vous connecter à chacune d'entre elles à chaque fois que vous le voulez simplement en fournissant son ID ("urlid") à SqlTool. Lorsque vous exécutez SqlTool de façon interactive il se comporte par défaut exactement comme vous le souhaitez. Si des erreurs se produisent, vous obtenez des messages d'erreur spécifiques et vous pouvez décider de revenir sur votre session. Vous pouvez facilement changer ce comportement pour auto-exécuter [auto-commit], sortir sur erreur, etc … , pour la session en cours ou pour toutes les invocations interactives. Vous pouvez importer ou exporter des fichiers de valeurs délimitées.

Lorsque vous exécutez SqlTool avec un script SQL, il se comporte par défaut exactement comme vous le souhaitez. Si une erreur est rencontrée, la connexion sera annulée, puis SqlTool quittera le programme en fournissant un code d'erreur. Si vous le souhaitez, vous pouvez détecter et traiter cette erreur (ou une autre) de la façon que vous voulez. Pour les scripts suspects de produire des erreurs (comme de nombreux scripts de base de données issus des éditeurs), vous pouvez continuer avec SqlTool-upon-erreur. Pour programmeurs de scripts SQL, vous aurez accès à des fonctionnalités portabilité de scripts qui vous n'avez jamais connues. Vous pouvez utiliser un jeu de variables en ligne de commande ou dans votre script. Vous pouvez traiter les erreurs spécifiques avec des commandes SQL ou vos variables. Vous pouvez enchaîner les scripts SQL, appeler des programmes externes, transférer des données à des fichiers et utiliser des déclarations préalables. Enfin, vous disposez d'un langage procédural avec les déclarations `if`, `foreach`, `while`, `continue` et `break`.

Plates-formes d'execution et versions de SqlTool

SqlTool s'exécute sur toute plate-forme Java à partir de la version 1.4. Je ne l'ai pas fait tourner avec une machine virtuelle Java non-Sun depuis des années (telle Blackdown, IBM, JRockit, etc), mais je n'ai eu aucune information faisant état de problèmes avec elles, SqlTool n'utilisant aucune classe propriétaire Sun directement. Certains des exemples ci-dessous utilisent des indications travaillant exactement comme dans une interface Unix compatible Bourne (seule l'indication de continuité de ligne devrait être changée pour les interfaces Unix compatibles C). Je n'ai pas encore testé ces commandes sous Windows et je doute que la citation fonctionnera telle quelle (même si cela est possible). SqlTool est encore un outil très utile, même si vous n'avez pas de capacité de tous les citer.

Si vous utilisez SqlTool d'une distribution HSQLDB avant la version 1.8.0.8, vous devez utiliser la documentation de cette distribution parce que ce manuel documente de nombreuses nouvelles fonctionnalités, des changements importants des commandes interactives et quelques modifications de compatibilité ascendante (voir la section suivante à ce sujet). Ce document est mis à jour pour les versions actuelles de SqlTool et SqlFile au moment où j'écris ceci (versions 333 et 354

respectivement, SqlFile est la catégorie qui fait le plus gros du travail pour SqlTool). Par conséquent, si vous utilisez une version de SqlTool ou SqlFile plus actualisée vous devrez trouver une nouvelle version de ce document. (L'imprécision est due au contenu indépendant des révisions successives et d'une ou deux corrections de bugs après la publication de la version). La bannière de démarrage affichera simultanément les versions lorsque vous lancerez SqlTool de façon interactive. (Les numéros des versions de SqlTool et SqlFile sont plus anciennes que les versions 333 et 354).

Ce guide couvre la versions de SqlTool livrée avec les séries 1.8 et 1.9 de HSQLDB.

Changements fonctionnels

Cette section répertorie les modifications apportées à SqlTool depuis la dernière version majeure de de HSQLDB laquelle peut assurer la portabilité des scripts SQL. Pour la révision de ce document, cette liste comporte les modifications apportées à SqlTool et ayant une influence sur les scripts après la sortie finale de la version 1.8.0.0 de HSQLDB. Je n'ai pas spécialement listé les changements des commandes uniquement interactives (les commandes ":" avec une exception d'héritage, lesquelles sont listées ci-dessous), puisque ces commandes ne peuvent être utilisées dans des scripts SQL. De plus, je n'ai pas spécialement listé les caractéristiques, additions et améliorations de compatibilité ascendante. La raison en est que lister toutes les améliorations serait très long.

- SqlTool traite maintenant correctement les sorties avec saut de ligne \r\n sur les plates-formes gérant le saut de ligne avec \r\n, comme Windows. Ceci inclut une sortie écrite vers stdout, les fichiers \w, et les fichiers \o. [SqlTool now consistently outputs \r\n line breaks when on \r\n-linebreak platforms, like Windows. This includes output written to stdout, \w files, and \o files]

- Les valeurs de type “temps” sont aussi bien gérées en tant que dates qu'en tant que temps. Ceci était rendu nécessaire face à la grande variété de formats présents dans les différentes bases de données.

- Les entrées DVSV utilisent désormais le format JDBC Timestamp avec les dates et, éventuellement l'heure de la journée.

- La commande “:” est maintenant une commande interactive stricte. Si vous voulez répéter une commande dans un script SQL, répétez simplement le texte exact de cette commande. Une utilisation non interactive est maintenant indépendante de l'historique des commandes.

- La commande “:w” remplace la commande “\w”. Contraitement à l'écriture en sortie vers un fichier avec \w, :w est utilisé pour écrire les commandes SQL d'une façon interactive.

- Les scripts shell (scrips de l'interpêteur de commandes) utilisant le mode brut [raw mode] (p.e. : PL/SQL scripts) doivent terminer le code brut avec une ligne contenant “.;”, laquelle enverra également le code d'execution à la base de données. (L'ancienne commande “.” a été changée en “:.” pour la faire clairement apparaître comme une commande interactive).

- L'argument –sql n'ajoutera jamais automatiquement une virgule au texte tapé. Si vous voulez executer une commande terminée par un demi-- alors tapez le.

Bien que cela n'affecte pas les scripts, je citerai un récent changement significatif des commandes interactives. Les commandes `Special` et `PL` ne sont pas conservés dans la mémoire tampon et dans l'historique des commandes, de sorte qu'ils peuvent être rappelés et modifiés comme les commandes SQL. Maintenant, seules les commandes `edit/history :` ne sont pas conservés dans le mémoire tampon et l'historique.

Connaissances minimales pour Exécuter SqlTool

ATTENTION

Si vous utilisez un serveur de base de données Oracle, il établira votre transaction courante si vous déconnectez proprement, sans savoir si vous avez configuré l'auto connexion ou non. Cela se produira si vous quittez SqlTool (ou tout autre client) de façon normale (par opposition à l'arrêt d'un processus ou par l'utilisation de Ctrl-C, etc. ...) Cela n'est mentionné dans cet article que par souci de concision, aussi je n'ai donc pas besoin de le mentionner dans le texte principal aux nombreux endroits où l'auto connexion est discutée. Ce comportement n'a rien à

voir avec SqlTool. Il s'agit d'une bizarrerie d'Oracle.

Si vous souhaitez utiliser SqlTool, alors soit vous avez un fichier texte SQL, soit vous tapez les commandes SQL de manière interactive. Si aucun de ces cas s'applique à vous, alors vous regardez le mauvais programme.

Procédure 8.1 pour executer SqlTool

1. Copiez le fichier `sqltool.rc` à partir du répertoire `sample` {1} de votre distribution HSQLDB vers le répertoire `home` et sécurisez son accès si votre ordinateur est accessible à d'autres personnes (notamment par l'intermédiaire d'un réseau). Ce fichier travaillera comme étant un processus de base de données uniquement en mémoire, ou si votre cible est un serveur HSQLDB en cours d'exécution sur votre ordinateur local avec les paramètres par défaut et le mot de passe pour le compte "sa" est vide (le mot de passe sa est vide lorsque de nouveaux objets de base de données HSQLDB sont créés). Éditez le fichier si vous avez besoin de changer l'URL du serveur cible, le nom d'utilisateur, le mot de passe, le jeu de caractères, le pilote JDBC, ou le TLS (Transport Layer Security = protocole de sécurisation de la couche de transport) tel que documenté dans la section Authentification de l'installation du fichier RC. (Vous pourriez, à défaut, utiliser la commande `--inlineRc` pour spécifier vos paramètres de connexion comme documenté dans la section Utiliser une authentification RC en ligne.
2. Découvrez où le fichier hsqldb.jar réside. En règle générale, il réside en HSQLDB_HOME`/lib/hsqldb.jar` où HSQLDB_HOME est le répertoire de base de votre installation du logiciel HSQLDB. Pour cette raison, je vais utiliser "$HSQLDB_HOME/lib/hsqldb.jar" comme chemin de `hsqldb.jar` pour mes exemples, mais comprenez que vous devez utiliser le bon chemin d'accès à votre propre fichier `hsqldb.jar`.
3. Executer

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

pour voir quels arguments en ligne de commande sont disponibles. Notez que vous n'avez pas à vous préoccuper de fixer le CLASSPATH lorsque vous utilisez le commutateur `-jar` pour java. En supposant que vous avez installé votre fichier SqlTool exécutable à l'emplacement par défaut et que vous voulez utiliser le pilote HSQLDB JDBC, vous voulez l'exécuter d'une façon comme :

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

pour une utilisation interactive, ou

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

ou

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

où `mem` est une urlid, et les arguments suivants sont des chemins de fichiers texte SQL. Pour les chemins de fichiers, vous pouvez utiliser n'importe quel caractère de remplacement supporté par votre système.

L'urlid `mem` dans ces commandes est une clé dans votre fichier RC (Run Commands), comme expliqué dans la section {RC File Authentification Setup}. Depuis d'elle est une base de donnée uniquement en mémoire, vous pouvez utiliser SqlTool avec cet urlid immédiatement, quelle que soit la configuration de la base de données (cependant, vous ne pouvez pas fixer chaque changement que vous apportez à cette base de données). L'échantillon de fichier sqltool.rc définit également l' urlid « localhost-sa » pour un serveur HSQLDB local. À la fin de cette section, je vais expliquer comment vous pouvez charger des échantillon de données pour jouer avec, si vous le souhaitez.

Important

SqlTool n'effectue pas les changements SQL par défaut. Cela permet à l'utilisateur de décider si ses digressions doivent être conservées ou révoquées. N'oubliez pas d'utiliser, soit run \= avant de quitter SqlTool (la plupart des bases de données supportent également la commande SQL `commit`, ou utilisez le commutateur `--autoCommit` en ligne de commande.

Si vous placez un fichier nommé `auto.sql` dans votre répertoire home, ce fichier sera exécuté automatiquement à chaque fois que vous exécutez SqlTool de façon interactive et sans le commutateur `--noAutoFile`.

Pour utiliser un pilote JDBC autre que le pilote HSQLDB, vous ne pouvez pas utiliser le commutateur `-jar`, car, alors, vous aurez besoin de modifier le classpath. Vous devez ajouter le fichier hsqldb.jar et votre pilote JDBC à votre classpath et vous devez dire à SqlTool ce quel pilote JDBC utiliser. Ce dernier peut être réalisé en utilisant le commutateur "--driver" , ou configurer la section « driver » de votre fichier de configuration. La section RC File Authentification Setup explique la deuxième méthode. Voici un exemple de la première méthode (après avoir fixé de façon appropriée le classpath).

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

Astuce

Si les tables de la requête sur votre écran sont en désordre du fait du gondolement des lignes, la meilleure et la plus facile des solutions est habituellement de redimensionner votre émulateur de terminal pour le rendre plus large. (Sous certaines conditions vous cliquez et faites glisser les bords du cadre pour redimensionner, autrement, vous utilisez un menu qui vous permet d'entrer le nombre de colonnes). }}

Si vous utilisez SqlTool pour vous connecter à un serveur de réseau HSQLDB ou de toute base de données non-HSQLDB, vous pouvez préférer utiliser le fichier jar `hsqltool.jar` ou `hsqldbutil.jar` à la place de `hsqldb.jar`. Ces fichiers jar contiennent tout SqlTool sans autre chose vous n'avez pas besoin, mais vous aurez à suivre une procédure simple pour générer ces fichiers jar. Voir la section Using hsqltool.jar and hsqldbutil.jar .

Types non affichables

Il existe de nombreux types SQL que SqlTool (basé sur un programme texte) ne peut afficher correctement. Cela inclut les typesSQL : `BLOB`, `JAVA_OBJECT`, `STRUCT`, et `OTHER`. Lorsque vous exécutez une requête qui retourne un de ces éléments, SqlTool va sauvegarder la toute première de ces valeurs obtenues dans la mémoire tampon binaire, et n'affichera pas toutes les résultats de cette requête. Vous pouvez alors sauvegarder la valeur binaire dans un fichier, comme expliqué dans la section Storing and retrieving binary files

Il existe d'autres types, tels que `BINARY`, que JDBC peut rendre affichables (en utilisant ResultSet.getString ()), mais que vous voudriez bien récupérer format binaire brut. Vous pouvez utiliser la commande \b pour récupérer chaque type de colonnes en format binaire brut (vous pouvez ensuite stocker la valeur dans un fichier binaire).

Il existe une autre restriction dans les clients de base de données de type texte, c'est leur incapacité pratique pour l'utilisateur de taper des données binaires telles que des photos, des flux audio, et des objets Java sériés. Vous pouvez utiliser SqlTool pour charger chaque objet binaire dans une base de données en disant à SqlTool d'obtenir insert/update des données à partir d'un fichier. Cela est aussi expliqué dans la section Storing and retrieving binary files.

Raccourcis

Les raccourcis et les icônes de lancement rapide sont utiles, en particulier si vous exécutez souvent SqlTool avec les mêmes arguments. Il est vraiment facile à mettre en place plusieurs d'entre eux - un pour chaque type d'invocation de SqlTool (par exemple, chacun devrait commencer SqlTool avec tous les arguments d'un démarrage typique). Une configuration type est d'avoir un raccourci pour chaque compte de base de données que vous utilisez normalement (utiliser un argument urlid différent pour chaque cible d'un raccourci).

Les spécifications des icônes varient en fonction de votre gestionnaire de votre bureau, bien sûr. Je

vais expliquer comment mettre en place une icône de démarrage de SqlTool sous Windows XP. Linux et Mac devraient être en mesure de le faire ainsi car il est plus facile de le faire avec un bureau sous Mac ou sous Linux.

Procédure 8.2. Création d'un raccourci sur le bureau pour SqlTool

1. Click droit dans la fenêtre principale
2. Nouveau (New)
3. Raccourci (Shortcut)
4. Parcourir (Browse=
5. Sélectionnez le répertoire où se trouve le JRE (Java Runtime Environment). Pour une installation récente le JRE de sun s'installe en `C:\ProgramFiles\Java\*\bin by default` (* étant le nom et le n° de version du JDK ou JRE)
6. Sélectionnez `java.exe`
7. OK
8. Suivant (Next)
9. Entrez un nom (Enter any name)
10. Terminer (Finish)
11. Clic droit sur la nouvelle icône.
12. Propriétés (Properties)
13. Editer la Cible (Edit the Target Field)
14. Laisser le chemin de java.exe exactement comme il est, y compris les guillemets, mais d'ajouter ce qi est nécessaire. Débutez par un espace puis entrez la ligne de commande que vous souhaitez exécuter.
15. Changer d'icône ... pour jolie icône.
16. Si vous voulez une icône de lancement rapide à la place de (ou en plus) d'une icône de raccourci sur le bureau, cliquez et faites la glisser vers votre barre de lancement rapide. (Vous pourrez avoir besoin ou non de modifier les propriétés de la barre d'outils Windows pour vous permettre d'ajouter de nouveaux éléments).

Charger des données exemples

Si vous voulez des exemples d'objets de base de données et vous amuser avec, exécutez le fichier SQL `sampledata.sql`. Ce fichier est localisé dans le répertoire `sample` de votre distribution HSQLDB [1]. Pour séparer les données exemples de vos données, vous pouvez la mettre dans votre propre schéma de fonctionnement en exécutant ceci avant de l'importer:

```
CREATE SCHEMA sampledata AUTORISATION dba;
SET SCHEMA sampledata;
```

Lancez-le ainsi à partir d'une session SqlTool

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

où HSQLDB_HOME est le répertoire d'installation de votre logiciel HSQLDB [1].

Pour les bases de données uniquement en mémoire, vous aurez besoin d'exécuter ceci à chaque fois que vous exécutez SqlTool. Pour les autres bases de données (persistantes), les données résident dans votre base de données jusqu'à ce que vous supprimiez les tables.

Authentification de l'installation du fichier RC

(RC File Authentication Setup)

Authentifier l'installation d'un fichier RC (Release Candidate) se fait en créant un fichier texte de configuration RC. Dans cette section, quand je dis configuration ou fichier de config, j'entends un fichier de configuration RC. Les fichiers RC sont utilisés par n'importe quel programme client JDBC utilisant la classe org.hsqldb.util.RCData-- Ceci inclut SqlTool, DatabaseManager, DatabaseManagerSwing. Vous pouvez également l'utiliser pour vos propres programmes client JDBC.

L'exemple suivant de fichier RC se localise ici : sample/sqltool.rc. Dans votre distribution HSQLDB [1].

Exemple 8.1. Échantillon de fichier RC

(Example 8.1. Sample RC File)

```
# $Id: sqltool.rc,v 1.22 2007/08/09 03:22:21 unsaved Exp $

# This is a sample RC configuration file used by SqlTool, DatabaseManager,
# and any other program that uses the org.hsqldb.util.RCData class.

# You can run SqlTool right now by copying this file to your home directory
# and running
#    java -jar /path/to/hsqldb.jar mem
# This will access the first urlid definition below in order to use a
# personal Memory-Only database.
# "url" values may, of course, contain JDBC connection properties, delimited
# with semicolons.

# If you have the least concerns about security, then secure access to
# your RC file.
# See the documentation for SqlTool for various ways to use this file.

# 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
# When connecting directly to a file database like this, you should
# use the shutdown connection property like this to shut down the DB
# properly when you exit the JVM.

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



# Template for a urlid for an Oracle database.
# You will need to put the oracle.jdbc.OracleDriver class into your
# classpath.
# In the great majority of cases, you want to use the file classes12.zip
# (which you can get from the directory $ORACLE_HOME/jdbc/lib of any
# Oracle installation compatible with your server).
# Since you need to add to the classpath, you can't invoke SqlTool with
# the jar switch, like "java -jar .../hsqldb.jar..." or
# "java -jar .../hsqlsqltool.jar...".
# Put both the HSQLDB jar and classes12.zip in your classpath (and export!)
# and run something like "java org.hsqldb.util.SqlTool...".

#urlid cardiff2
#url jdbc:oracle:thin:@aegir.admc.com:1522:TRAFFIC_SID
#username blaine
#password secretpassword
#driver oracle.jdbc.OracleDriver



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

#urlid tls
#url jdbc:hsqldb:hsqls://db.admc.com:9001/lm2
#username blaine
#password asecret
#truststore /home/blaine/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
#username blaine
#password hiddenpwd
#driver com.mysql.jdbc.Driver

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

# Template for a Microsoft SQL Server database
#urlid msprojsvr
#url jdbc:microsoft:sqlserver://hostname;DatabaseName=DbName;SelectMethod=Cursor
# The SelectMethod setting is required to do more than one thing on a JDBC
# session (I guess Microsoft thought nobody would really use Java for
# anything other than a "hello world" program).
# This is for Microsoft's SQL Server 2000 driver (requires mssqlserver.jar
# and msutil.jar).
#driver com.microsoft.jdbc.sqlserver.SQLServerDriver
#username myuser
#password hiddenpwd

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

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

Vous pouvez enregistrer ce fichier ou bon vous semble, et spécifier l'adresse dans SqlTool/DatabaseManager/DatabaseManagerSwing en utilisant l'argument `--rcfile`. S'il n'y a pas de raison de ne pas utiliser l'adresse par défaut (et il y a des situations ou vous ne voudrez pas), utilisez l'adresse proposée et vous n'aurez pas à donner l'argument `--rcfile` à SqlTool/DatabaseManager /DatabaseManagerSwing. L'adresse par défaut est sqltool.rc ou dbmanager.rc dans votre dossier racine (celui du programme qui l'utilise). Si vous avez un doute sur l'adresse de votre dossier racine, exécutez SqlTool avec un urlid de téléphone (? - with a phony urlid)et il vous dira ou il s'attend à trouver le fichier de configuration.

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

Le fichier de configuration consiste en une déclaration de ce type:

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

Ces quatre réglages sont requis pour toute urlid. (Il y a aussi des réglages optionnels, qui sont décrits un peu plus bas). L'URL doit contenir les propriétés de la connexion JDBC. Dans le fichier, vous pouvez avoir autant de lignes vierges et de commentaires comme :

```
   # This comment
```

que vous voulez. Le fait est que l'urlid que vous donnez dans votre commande SqlTool/DatabaseManager doit concorder avec une urlid de votre fichier de configuration.

Important

Utilisez toutes les opportunités à votre disposition pour protéger votre fichier de configuration.

Il doit être lisible, à la fois localement et à distance, seulement pour les utilisateurs qui exécutent des programmes qui en ont besoin. Sous UNIX, c'est facile à résoudre par les commandes chmod/chown et en s'assurant qu'il est protégé des accès anonymes à distance (par ex. via NFS, FTP ou Samba).

Vous pouvez également spécifier les réglages optionnels suivants dans une déclaration urlid. Le réglage sera seulement, bien sur, appliqué à cet urlid.

charset
: Utilisé par le programme SqlTool, mais pas par le programme gestionnaire de base de données. Voyez la section Encodage des caractères dans la section non-interactive. Vous pouvez alternativement appliquer ce réglage pour une invocation SqlTool en définissant la propriété système sqlfile.charset. La valeur par défaut est US-ASCII.

driver
: Définit le nom de classe du pilote JDBC. Vous pouvez alternativement appliquer ce réglage pour une invocation SqlTool/DatabaseManager en utilisant l'interrupteur en ligne de commande --driver. La valeur par défaut est org.hsqldb.jdbcDriver.

truststore
: TLS fait confiance au chemin de stockage des trousseaux de confiance comme documenté dans le chapitre TLS. Vous n'aurez généralement besoin de définir cela que si le serveur utilise un certificat non publiquement certifié (par ex. un certificat auto-signé auto-CA).

La propriété et les interrupteurs en ligne de commande SqlTool prévalent sur les réglages effectués dans le fichier de configuration.

Utiliser une authentification RC en ligne

(Using Inline RC Authentication)

L'installation d'une authentification RC en ligne s'effectue en utilisant l'interrupteur en ligne de commande --inlineRc sur SqlTool. L'interrupteur en ligne de commande --inlineRc traite une liste d'éléments Clé/Valeur séparés par une virgule. Les éléments URL et Utilisateur sont requis. Le reste est optionnel.

url
: L'URL JDBC de la base de données à laquelle vous voulez vous connecter.

user
: Le nom de l'utilisateur qui se connecte à la base de données.

charset
: Défini l'encodage des caractères. Par défaut à US-ASCII.

trust
: Le chemin du trousseau de confiance comme documenté dans le chapitre TLS.

password
: Vous ne devez utiliser cet élément que pour définir un mot de passe vierge, comme `password=`. Pour toute autre valeur de mot de passe, omettez l'élément mot de passe, le système vous demandera la valeur.

(Utilisez l'interrupteur --driver au lieu de --inlineRc pour spécifier une classe de pilote JDBC). Voici un exemple d'invocation de SqlTool pour se connecter à une base de données **standalone**.

```
java -jar $HSQLDB_HOME/lib/hsqldb.jar
     --inlineRc URL=jdbc:hsqldb:file:/home/dan/dandb,USER=dan
```

Pour des raisons de sécurité, vous ne pouvez pas spécifier un mot de passe non vide comme argument. C'est dans le processus de connexion que vous devrez renseigner le mot de passe.

Utiliser la version courante de SqlTool avec une distribution HSQLDB plus ancienne

(Using the current version of SqlTool with an older HSQLDB distribution.)

Cette procédure permet aux utilisateurs d'une version héritée (legacy version) de HSQLDB d'utiliser toutes les nouvelles fonctionnalités de SqlTool. Vous obtiendrez également les nouvelles versions des gestionnaires de bases de données ! Cette procédure fonctionne pour les distributions remontant au moins jusqu'à la 1.7.3.3, et probablement plus anciennes encore.

Suivez les instructions de la section Utilisation de hsqltool.jar et hsqldbutil.jar pour construire le fichier jar `hsqldbutil.jar`.

A partir de maintenant, quand vous allez lancer SqlTool, assurez vous d'avoir ce `hsqldbutil.jar` comme première item dans votre CLASSPATH. Vous ne pouvez pas exécuter Sqltool avec l'interrupteur "-jar" (parce que l'interrupteur -jar ne permet pas de définir votre propre chemin de classe).

Voici un exemple UNIX où quelqu'un voudrait utiliser le nouvel SqlTool avec ses ancienne bases de données, aussi bien qu'avec Postgresql et une application locale.

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

Usage interactif

(Interactive Usage)

Veuillez s'il vous plaît Lire la section Le strict minimum avant de lire celle-ci.

Vous lancerez SqlTool de façon interactive en ne spécifiant pas de chemin SQL sur la ligne de commande SqlTool. Comme ceci :

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

Procédure 8.3 Que se passe t-il quand SqlTool est exécuté interactivement (en utilisant tous les réglages par défaut)

(Procedure 8.3. What happens when SqlTool is run interactively (using all default settings))

1. SqlTool démarre et se connecte à la base de données spécifiée, en utilisant votre fichier de configuration SqlTool (comme expliqué dans la section Authentification de l'installation du fichier RC).
2. Le fichier SQL auto.sql de votre répertoire racine est exécuté (s'il existe).
3. SqlTool affiche une bannière montrant les numéros de version de SqlTool et de SqlFile, et décrit les différents types de commandes que vous pouvez envoyer aussi bien que les commandes listant toutes les commandes spécifiques disponibles.

Vous quitterez votre session par la commande spéciale "\q" ou un raccourci de fermeture (ending input) (comme avec Ctrl-D ou Ctrl-Z)

Important

Chaque commande (quel que soit le type) et commentaire doit commencer en début de ligne (ou immédiatement après la fin d'un commentaire avec "*/").

Vous ne pouvez pas encapsuler (nest) de commandes ni de commentaires. Vous pouvez seulement démarrer de nouvelles commandes (et commentaires) une fois la déclaration

précédente terminée. (Rappelez vous que si vous exécutez SqlTool de manière interactive, vous pouvez terminer une déclaration SQL sans l'avoir exécutée en entrant une ligne vierge).

(Les commandes spéciales, celles d'édition du tampon (Edit Buffer Commands) et les commandes PL consistent toujours en une seule ligne. Chacune de ces commandes ou commentaires doit être précédée de caractères espace.)

Ces règles ne s'appliquent pas du tout au mode Raw. Le mode Raw est réservé aux utilisateurs avancés quand ils veulent complètement désactiver SqlTool de façon a pouvoir transmettre directement de gros blocs de texte au moteur de la base de données.

Types de commande

(Command Types)

Quand vous tapez dans SqlTool, vous tapez toujours des parties de la **commande immédiate**. Vous exécutez la commande immédiate en pressant ENTER après un point virgule (pour les commandes SQL) ou juste en pressant ENTER (toute autre commande non-vide-- voyez la section prochaîne pour cette distinction). L'interactivé : Les commandes peuvent effectuer des action avec ou sur le tampon d'édition. **Le tampon d'édition** contient généralement une copie de la dernière commande exécutée, et vous pouvez la visualiser en permanence avec la commande :b. Si vous n'utilisez jamais de commande ":", vous pouvez complètement ignorer le tampon d'édition. Si vous voulez répéter des commandes ou editer des commandes précédentes, vous aurez besoin de travailler avec le tampon d'édition. La commande immédiate contient exactement tout ce que vous tapez. L'historique de commande et le tampon d'édition peut contenir tout type de commande autre que les commentaires et les commandes ":" (c.a.d. que les commandes ":" et les commentaires ne sont tout simplement pas copiés dans l'historique ou dans le tampon d'édition).

Heureusement un exemple va clarifier la différence entre la commande immédiate et le tampon d'édition. Si vous tapez dans le tampon d'édition la commande Substitution "`:s/tbl/table/`", la commande :s que vous avez tapé est la commande immédiate (et ne sera jamais stocké dans le tampon d'édition ou historique, puisque c'est une commande ":"), mais le but de la commande de substitution est de modifier les contenus du tampon d'édition (Effectuez une substitution sur ce dernier)-- le but étant qu'après vos substitutions vous exécutiez le tampon avec la commande ":;". La commande ":a" est spéciale en ce sens que lorsque vous pressez ENTER pour l'exécuter, elle copie le contenu du tampon d'édition dans une nouvelle commande immédiate et vous laisse dans un état ou vous allez compléter cette commande immédiate (presque) exactement comme si vous veniez de la taper.

Types de commande

(Command Types)

Types de commande

Note

Au dessus, nous disions que si vous entriez une commande SQL, elle correspondait à une commande SqlTool. C'est l'utilisation la plus courante, toutefois, vous pouvez en fait mettre de multiples déclarations SQL dans une commande SQL. Un exemple serait :

```
INSERT INTO t1 VALUES(0); SELECT * FROM t1;
```

C'est une commande SqlTool contenant deux déclarations SQL. Lisez la section Découpage (Chunking) pour comprendre pourquoi vous voudriez découper des commandes SQL, comment, et ce que cela implique.

Déclaration SQL

Toute commande que vous entrez qui ne commence pas par "\", ":", ou "* " est une déclaration SQL. La commande n'est pas terminée quand vous pressez ENTER, comme dans la plupart des shells de système d'exploitation. Vous terminez les déclarations SQL soit avec ";" à la fin de la ligne, ou avec une ligne vierge. Dans le premier cas (In the former case), la déclaration SQL sera exécutée sur (against) la base de données SQL, la commande ira dans le tampon d'édition et l'historique de commande SQL pour l'éditer ou la voir plus tard. Dans le premier cas, exécuter sur la base de données SQL signifie transmettre le texte SQL au moteur de base de données pour exécution. Dans le dernier cas (vous terminez une déclaration SQL avec une ligne vierge), la commande ira dans le tampon d'édition et l'historique SQL, mais ne sera pas exécutée (vous pourrez cependant l'exécuter plus tard à partir du tampon d'édition). (Voir la note immédiatement ci-dessus à propos de multiples déclarations SQL dans une commande SqlTool).

(Les lignes vierges ne sont interprétées de cette façon que lorsque SqlTool est exécuté de manière interactive. Dans les fichiers SQL, les lignes vierges dans les déclarations SQL restent une partie de la déclaration SQL).

En conséquence de ces règles de terminaison, dès que vous entrez du texte qui ne soit pas une commande spéciale, tampon d'édition / historique, ou commande PL, vous ajoutez toujours des lignes à une déclaration SQL ou un commentaire. (Dans le cas de la première ligne, vous ajouteriez à une déclaration SQL vide. Par exemple vous démarreriez une nouvelle déclaration SQL ou un nouveau commentaire).

Commande spéciale

Exécutez la commande "\?" pour obtenir la liste des commandes spéciales. Elles commencent toutes par "\". Je décrirais quelques unes des commandes spéciales les plus pratiques plus bas.

Commande Tampon d'édition / Historique

(Edit Buffer / History Command)

Exécutez la commande ":?" pour lister les commandes Tampon d'édition / Historique. Toutes ces commandes commencent par ":". Ces commandes utilisent des commandes de l'historique de commande, ou opèrent sur le tampon d'édition, pour que vous puissiez éditer et/ou (ré)exécuter des commandes précédemment entrées.

Commande PL

Commandes de langage procédural. (PL = Procedural Langage). Exécutez la commande "*?" pour afficher la liste des commandes PL. Toutes les commandes PL commencent par "*". Les commandes PL servent à la déclaration et l'utilisation de variables de script, conditionnelles et aux déclarations de contrôle de flux comme `* if` et `* while`. Quelques fonctionnalités PL (telles que les alias PL, mettre à jour et sélectionner les données directement de/vers les fichiers) peuvent être d'une réelle utilité pour pratiquement tous les utilisateurs, et seront donc décrites brièvement dans cette section. Une explication plus détaillée des variables et d'autres fonctionnalités PL, avec des exemples, est développée dans la section Langage procédural de SqlTool.

Mode Raw

(Raw Mode)

Les descriptions d'écriture de commande ci-dessus ne s'appliquent pas au mode Raw (Raw = Brut). Dans le mode Raw, SqlTool n'interprète plus du tout ce que vous tapez. Ces entrées ne font qu'aller dans le tampon d'édition que vous pouvez envoyer au moteur de la base de données. Les débutants prendront moins de risques en ignorant le mode Raw. Vous ne le rencontrerez jamais à moins que vous entriez la commande spéciale "\.", ou une commande PL/SQL. Pour plus de détails, voyez la section .

Commandes spéciales

(Special Commands)

Commandes spéciales essentielles

\?	Aide

\q	Quitter
\i chemin/vers/script.sql	Exécute le script SQL spécifié, puis reprend le comportement interactif.
\=	Valide la transaction SQL en cours. La plupart des utilisateurs ont l'habitude d'entrer la déclaration SQL `commit;`, mais cette commande est critique pour ces bases de données qui ne prennent pas en compte la déclaration. Ce n'est évidemment plus nécessaire si vous avez activé le mode auto-commit.
\x?	Affiche le sommaire de l'export DSV, et toutes les options DSV disponibles.
\m?	Affiche le sommaire de l'import DSV, et toutes les options DSV disponibles.
\d?	**Affiche le sommaire des commandes \d ci-dessous :** \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] **Listes des objets disponibles par type** (Lists available objects of the given type.) ■ t: Tables non système ■ v: Vues ■ s: Séquences ■ i: Index ■ S: Tables Système ■ a: Alias ■ n: Noms de schéma ■ u: Utilisateurs de base de données ■ r: Rôles ■ *: Tous les objets ressemblant à des tables Si votre base de données prend en charge les schémas, le nom du schéma apparaitra également. Si vous fournissez un filtre optionnel sous forme d'une sous-chaine, alors seulement les items contenant la sous-chaine donnée (dans le nom d'objet ou le nom du schéma) seront listées.

Important

La sous-chaine test est sensible à la casse ! Bien que dans les requêtes SQL et pour la commande "\d objectname" les noms d'objets soient habituellement insensibles à la casse, pour les commandes \dx, vous devez formater la sous-chaine filtre exactement comme elle apparait dans l'affichage (output) special command. C'est un inconvénient, puisque le moteur de la base de données

changera les noms dans SQL dans la casse par défaut à moins que vous n'encadriez le nom de guillemets, mais c'est une fonctionnalité côté serveur qui ne peut pas (portabilité) être reproduite par SqlTool. Vous pouvez utiliser des espaces et autres caractères spéciaux dans la chaine.

Astuce

Les sous-chaines de filtre terminant par un "." sont spéciales. Le point signifie réduire la recherche au nom de schéma donné (sensible à la casse). Par exemple, si j'exécute "\d* BLAINE.", ceci listera tous les objets du "genre" table dans le schéma "BLAINE". La mise en capitales du schéma doit être exactement la même que l'affichage du nom de schéma obtenu par la commande "\dn". Vous pouvez utiliser des espaces et autres caractères spéciaux dans la chaine. (Par exemple, entrez le nom exactement comme vous l'entreriez entre guillemets dans une commande SQL). C'est un inconvénient, puisque le moteur de la base de données changera les noms dans SQL dans la casse par défaut à moins que vous n'encadriez le nom de guillemets, mais c'est une fonctionnalité côté serveur qui ne peut pas (portabilité) être reproduite par SqlTool.

Important

Les index ne doivent pas être recherchés par des sous-chaines, mais seulement par le nom exact de la table cible. Donc si I1 est un index de la table T1, vous listerez cet index en exécutant "\di T1". De plus, beaucoup de vendeurs de bases de données feront un rapport d'index seulement si une table cible est identifiée. Par conséquent, la commande "\di" sans argument n'aboutira pas si le vendeur de la base de données ne le prend pas en charge.

Affiche les noms des colonnes de la table ou vue spécifiée. NomObjet doit être un nom de table ou un nom schema.object.

Si vous fournissez une chaine filtre, seulement les colonnes dont le nom contient le filtre donné seront affichées. Le nom d'objet est pratiquement toujours insensible à la casse (dépend de votre base de données), mais le filtre, lui, est toujours sensible à la casse. Vous trouverez un côté pratique indéniable à ce filtre comparativement à d'autres utilitaires de base de données, ou vous ne pouvez que lister toutes les colonnes de grandes tables quand vous n'êtes intéressé que par l'une d'entre elles.

\d NomObjet [filter]

Astuce

En travaillant avec des données réelles (à l'opposé d'apprendre ou jouer), j'ai souvent trouvé pratique d'exécuter deux sessions SqlTool dans deux fenêtres de terminal cote à cote. J'effectue tout le travail réel (effectif) dans une fenêtre, et utilise l'autre principalement pour les commandes \d. De cette façon je peux me référer au dictionnaire des données pendant l'écriture de commandes SQL, sans avoir à scroller.

La liste présente n'inclut que les commandes spéciales essentielles, mais notez bien qu'il y a d'autres commandes spéciales pratiques que vous pouvez lister en exécutant \?. (Vous pouvez, par exemple, exécuter le SQL depuis des fichiers externes SQL, et sauvegarder vos commandes SQl interactives dans des fichiers). Quelques commandes spécifiques de ces autres commandes sont spécifiées immédiatement ci-dessous, et la section Génération de texte ou de rapports HTML explique comment utiliser les commandes spéciales "\o" et "\H" pour générer des rapports.

Soyez attentif au fait que la commande spéciale \! ne fonctionne pas pour des programmes externes qui ne lisent que depuis l'entrée standard. Vous pouvez faire appel à des programmes non-interactifs et interactifs graphiques, mais pas des programmes interactifs en ligne de commande.

SqlTool exécute les programmes \! directement, il ne lance pas un shell du système d'exploitation (ceci pour éviter du code spécifique à un système d'exploitation dans SqlTool). Par le fait, vous pouvez donner autant d'arguments en ligne de commande que vous souhaitez, mais vous ne pouvez pas utiliser les jokers du shell ou la redirection.

La commande \w peut servir à stocker chaque commande de votre historique SQL dans un fichier. Restaurez simplement la commande dans le tampon d'édition avec une commande comme "\-4" avant d'envoyer la commande \w.

Tampon d'édition / Commandes d'historique

(Edit Buffer / History Commands)

Tampon d'édition / Commandes d'historique

`:?`	Aide
`:b`	Liste le contenu courant du tampon d'édition.
`:h`	Montre l'historique des commandes. Pour chaque commande qui a été exécutée (jusqu'à la longueur maximum admissible de l'historique), l'historique de commande SQL montrera la commande ; son numéro de commande (#) ; et également de combien de commandes en arrière est sa position (un nombre négatif). Les commandes ":" ne sont jamais ajoutées à la liste de l'historique. Vous pouvez alors utiliser soit la forme de l'identifiant de la commande pour rappeler une commande dans le tampon d'édition (la prochaine commande décrite) ou comme la cible de chacune des commandes ":" suivantes. Cette dernière méthode est réalisée selon une manière très similaire à l'éditeur vi. Vous spécifiez le numéro de la commande cible entre la colonne et la commande. En guise d'exemple, si vous avez donné la commande :s/X/Y/, ceci effectuera la substitution du contenu du tampon d'édition ; mais si vous avez donné la commande :-3 s/X/Y/, ceci fera remonter la commande 3 dans l'historique de commande (et copiera la sortie dans le tampon d'édition). De même, exactement comme vi, vous pouvez identifier la commande à rappeler par l'utilisation d'une expression régulière à l'intérieur de slashs, comme :/bleu/ s/X/Y/ pour opérer sur la dernière commande que vous avez exécuté qui contenait "bleu".
`:13` OU `:-2` OU `:/blue/`	Rappelle une commande de l'historique de commande dans le tampon d'édition. Entrez ":" suivi du numéro positif de la commande de l'historique de commande, comme ":13"... ou ":" suivi par un nombre négatif, comme ":-2" pour revenir deux commandes en arrière dans l'historique de commande... ou ":" suivi par une expression régulière encadrée de slashs, comme ":/bleu/" pour rappeler la dernière commande contenant "bleu". La commande spécifiée sera écrite dans le tampon d'édition pour que vous puissiez l'exécuter ou l'éditer en utilisant les commandes ci-dessous. Comme décrit pour la commande :h immédiatement ci-dessus, vous pouvez ici faire suivre le numéro de commande par une des commandes ci-dessous pour réaliser l'opération donnée sur la commande spécifiée depuis l'historique plutôt que dans le contenu du tampon d'édition. Ainsi, par exemple, ":4;" chargera la commande 4 de l'historique puis l'exécutera (voyez la commande ":;" ci-dessous).
`:;`	Exécute la déclaration SQL, spéciale ou PL dans le tampon d'édition (par défaut). C'est une commande extrêmement pratique. Il est facile de s'en rappeler parce qu'elle consiste en ":", signifiant la commande du tampon d'édition, plus une terminaison de ligne ";", (qui signifie généralement l'exécution d'une déclaration SQL, bien que dans ce cas elle exécute également une commande spéciale ou PL).
`:a`	Entre dans le mode ajout (append) avec le contenu du tampon d'édition (par défaut) comme la commande courante. Quand vous pressez ENTER, les choses se passeront exactement comme si vous aviez physiquement retapé la commande qui est dans le tampon d'édition. Toute ligne que vous entrerez ensuite sera ajoutée à la commande

immédiate. Comme toujours, vous avez le choix entre presser ENTER pour exécuter une commande spéciale ou PL, entrer une ligne vierge pour stocker la commande dans le tampon d'édition, ou finir une déclaration SQL par le point virgule suivi de ENTER pour l'exécuter.

Vous pouvez, optionnellement, entrer une chaine après le :a, auquel cas les choses seront exactement comme décrites excepté que le texte additionnel sera aussi ajouté à la nouvelle commande immédiate. Si vous entrez une chaine après le :a terminant par ";", alors la commande immédiate résultante sera directement exécutée, comme si vous aviez tapé et confirmé la chaine entière.

Si votre tampon d'édition contient SELECT x FROM maTab et que vous exécutez a:le, la commande résultante sera SELECT x FROM maTable. Si votre tampon d'édition contient SELECT x FROM maTab et que vous exécutez a: ORDER BY y, la commande résultante sera SELECT x FROM maTab ORDER BY y. Notez que dans le dernier cas le texte ajouté commence par le caractère espace.

La commande de substitution est la première méthode pour l'édition de commande de SqlTool-- elle opère par défaut sur le tampon d'édition courant. Les arguments "to string" et "switches" sont tous deux optionnels (bien que le "/" final ne le soit pas). Pour commencer, je discuterais de l'utilisation et du comportement si vous ne fournissez aucun interrupteur du mode de substitution.

N'utilisez pas "/" s'il intervient dans une des deux chaines "from string" ou "to string". Vous pouvez utiliser le caractère de votre choix à la place du"/", mais il ne doit pas figurer dans une des deux chaines "from" ou "to". Exemple :

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

La chaine "to" est substituée pour la première occurence (cas spécifique) de la chaine "from". Le remplacement considèrera la déclaration SQL dans son ensemble, même si c'est une déclaration multi-lignes.

`:s/from regex/to string/switches`

Dans l'exemple ci-contre, le "from regex" est une chaine non formatée (plain string), mais elle est interprétée comme une expression régulière de façon à ce que vous puissiez bénéficier de toute la puissance des substitutions. Consultez la page man `perlre`, ou la spécification API java.util.regex.Pattern (http://http//java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html) pour tout ce que vous avez besoin de connaitre sur les expressions régulières étendues.

Ne terminez pas une chaine "to" par ";" dans l'intention d'obtenir une exécution de commande. Il y a un interrupteur de mode de substitution à utiliser dans ce but.

Vous pouvez utiliser toute combinaison d'interrupteur de mode de substitution.

- Utilisez "i" pour rendre les recherches pour "from regex" insensibles à la casse.
- Utilisez "g" pour remplacer tout, par exemple remplacer toutes les occurences de "from regex" plutôt que la première trouvée.
- Utilisez ";" pour exécuter la commande immédiatement après que le remplacement ait eu lieu.
- Utilisez "m" pour ^ et $ pour vérifier chaque saut de ligne dans un tampon d'édition multi-lignes, plutôt que seulement au tout début et à la toute fin du tampon entier.

Si vous spécifiez un numéro de commande (depuis l'historique de commande), vous aboutirez à une fonctionnalité réminiscente de vi, mais encore plus puissante, car les expressions régulières Perl/Java sont un "superset" des expressions régulières de vi. En guise d'exemple,

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

démarrera avec le numéro de commande 24 depuis l'historique de commande,

	remplacera par "needle" toutes les occurences de "pin", puis exécutera le résultat de ce remplacement (et cette déclaration finale sera bien sur copiée dans le tampon d'édition et l'historique de commande).
`:w /chemin /vers/fichier.sql`	Ceci ajoute le contenu du tampon courant (par défaut) dans le fichier spécifié. Puisque les lignes écrites sont des commandes Spécial, PL ou SQL, vous êtes effectivement en train de créer un script SQL.

Je trouve les constructions ":/regex/" et ":/regex/;" particulièrement pratiques pour un usage quotidien.

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

Ré-exécute la dernière commande \d que vous aviez donné (Le deuxième "\" est nécessaire pour oter la signification spéciale de "\" dans les expressions régulières). C'est merveilleux de pouvoir rappeler et exécuter la dernière commande "insert", par exemple, sans avoir besoin de vérifier l'historique ou garder la mémoire de combien de commandes en arrière c'était. Pour ré-exécuter la dernière commande d'insertion, lancez seulement ":/insert/;". Si vous voulez ne pas prendre de risques, faites le en deux temps pour vérifier que vous n'avez pas accidentellement rappelé une autre commande qui se trouve contenir la chaine "insert", comme

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

(Exécution de la dernière commande seulement quand vous êtes certain de la commande que SqlTool a restaurée). Souvent, bien sur, vous voudrez changer la commande avant de la ré-exécuter, et c'est là que vous aurez besoin de la combinaison des commandes :s et :a.

Nous finirons par quelques points subtils des commandes d'édition/de tampon. Vous ne pouvez généralement pas utiliser des variables PL dans les commandes d'édition/de tampon, pour éliminer les ambiguïtés et complexités possibles lors de la modification de commandes. La commande :w est une exception à cette règle, puisqu'elle peut être pratique pour utiliser des variables déterminant le fichier de sortie, et puisque cette commande ne fait aucun type d'édition.

L'aide `:?` explique comment vous pouvez changer le comportement de vérification d'une expression régulière par défaut (sensibilité à la casse, etc.), mais vous pouvez toujours utiliser une syntaxe comme "(?i)" à l'intérieur de votre expression régulière, comme décrit dans les spécifications API Java pour la classe java.util.regex.Pattern, trouvée ici (http://http//java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html). La vérification des commandes de l'historique (History-command-matching) avec la construction /regex/ est intentionnellement libérale, vérifiant chaque portion de la commande, sensible à la casse, etc., mais vous pouvez encore utiliser la méthode qui vient d'être décrite pour modifier ce comportement. Dans ce cas, vous pouvez utiliser "(?-i)" au début de votre expression régulière qui deviendra sensible à la casse.

Commandes PL

(PL Commands)

Voir sur Wikipédia : Langage Procédural (http://fr.wikipedia.org/wiki/PL/SQL)

L'essentiel de la commande PL (Essential PL Command)

* VARNAME = value	Définie la valeur d'une variable. Si la variable n'existe pas encore, elle sera créée. L'usage le plus commun est tel que vous pouvez l'utiliser plus tard dans des déclarations SQL, déclarations d'impression, et conditions PL, par utilisation des constructions *{VARNAME} ou *{:VARNAME}. La seule différence entre *{VARNAME} et *{:VARNAME} est que la première produit une erreur si VARNAME n'est pas définie, alors que la dernière développera une chaine de caractères de longueur zéro si VARNAME n'est pas définie.

Si vous définissez une variable dans une déclaration SQL (sans le ";" final) vous pouvez alors l'utiliser comme un alias PL tel que /VARNAME, dans cet exemple.

Exemple 8.2. Définition et utilisation d'un alias PL (variable PL)

```
* qry = SELECT COUNT(*) FROM MaTable
\p La requête stockée est '*{qry}'
/qry;
/qry WHERE mass > 200;
```

Si vous écrivez des définitions de variables dans le fichier SQL auto.sql dans votre dossier racine, ces alias/variables seront toujours disponibles pour l'utilisation interactive.

Les variables PL peuvent être développées dans toute commande autre que les commandes d'édition / de l'historique.

* load VARNAME /file/path.txt	Définie VARNAME avec le contenu du fichier ASCII spécifié.
* prepare VARNAME	Indique que la prochaine commande devrait être une commande SQL INSERT ou UPDATE contenant un point d'interrogation. La valeur de VARNAME sera substituée par la variable "?". Ceci fonctionne également pour les colonnes CLOB.
* VARNAME _	Quand la commande SQL suivante est exécutée, au lieu d'afficher les lignes, enregistre seulement la valeur de la toute première colonne dans la variable VARNAME. Ceci fonctionne pour les colonnes CLOB aussi. Ceci fonctionne également avec les colonnes Oracle type XML si vous utilisez les étiquettes de colonne et la fonction getclobval.
* VARNAME ~	Exactement la même que `* VARNAME ~` excepté que les résultats retournés seront affichés en plus de définir la variable.
* dump VARNAME /file/path.txt	Stocke la valeur de VARNAME dans le fichier ASCII spécifié.

Variable "?"

(? Variable)

Vous ne définissez pas la variable "?". Elle est simplement comme la variable "?" du Bourne shell (http://fr.wikipedia.org/wiki/Bourne_shell) en ce sens qu'elle est toujours automatiquement définie à la première valeur d'un jeu de résultat (ou la valeur retournée par d'autres commandes SQL). Elle fonctionne exactement comme la commande

```
* VARNAME ~
```

décrite ci-dessus, mais elle arrive automatiquement. Vous pouvez, bien sur, **dereference** "?", comme toute variable PL... described above, but it all happens automatically. You can, of course, dereference ? like any PL variable, **but it does not list with the**

```
list
```

et

```
listvalues
```

commands. You can see the value whenever you want by running

```
\p  *{?}
```

Remarquez bien que les commandes PL sont utilisées pour importer et télécharger des valeurs de colonnes depuis / vers des fichiers ASCII locaux, mais que les actions correspondantes pour les fichiers binaires utilisent les commandes spéciales \b. Ceci parce que les variables PL sont utilisées pour des valeurs ASCII et que vous pouvez stocker plusieurs valeurs de colonnes dans les variables PL. Ce n'est pas vrai pour les valeurs de colonnes binaires. La commande \b fonctionne **with a single binary byte buffer**.

Reportez vous à la section Langage procédural de SqlTool ci-dessous pour d'autres façons d'utiliser les variables, et de l'information sur les autres commandes et fonctionnalités PL.

Stocker et récupérer les fichiers binaires

(Storing and retrieving binary files)

Vous pouvez importer (upload) des fichiers binaires comme des photographies, des fichiers audio, ou des objets Java sérialisés (serialized ?) dans des colonnes de bases de données. SqlTool réserve un tampon binaire avec lequel vous pouvez charger depuis le fichier avec la commande \bl, ou depuis une requête à la base de données, en effectuant une requête d'un ligne pour chaque type non affichable (incluant BLOB, OBJECT, et OTHER). Dans le dernier cas, les données retournées pour la première colonne non affichable de la première ligne de résultats seront stockées dans le tampon binaire.

Quand vous avez les données dans le tampon binaire, vous pouvez les importer dans une colonne de base de données (incluant les types de colonne BLOB, OBJECT, et OTHER), ou les sauvegarder dans un fichier. La première action est accomplie par la commande spéciale \bp suivie d'une requête SQL préparée contenant un "réservoir" point d'interrogation (containing one question mark place-holder to indicate) pour indiquer ou les données doivent être insérées. La deuxième action s'effectue par la commande \bd.

Vous pouvez également stocker la valeur d'une colonne normale (affichable) dans le tampon binaire par l'utilisation de la commande spéciale \b. La toute première valeur de colonne de la première ligne de résultats de la commande SQL qui suivra sera stockée dans le tampon binaire octal. (binary byte buffer.)

Exemple 8.3. Insertion de données binaires dans une base de données depuis un fichier

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

Exemple 8.4. Télécharger des données binaires depuis une base de données vers un fichier

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

Vous pouvez également stocker et récupérer des valeurs de colonnes de texte depuis / vers des fichiers ASCII. Voir la section Commandes PL.

Historique des commandes

Command History

L'historique SQL accessible depuis la commande \h, et utilisé par d'autres commandes, est limité aux cent premières entrées, puisque son utilité vient du fait de pouvoir afficher rapidement la liste de l'historique. Vous pouvez changer la longueur de l'historique en définissant la propriété système property sqltool.historyLength comme un entier tel que :

```
java -Dsqltool.historyLength=100 -jar $HSQLDB_HOME/lib/hsqldb.jar urlid
```

S'il y avait quelque demande, je pourrai simplifier l'accès à ce réglage.

La liste de l'historique SQL contient toutes les commandes exécutées autres que celles du tampon d'édition et les commentaires, même si la commande contient une erreur de syntaxe ou se résout par un échec. La raison d'inclure les mauvaises commandes est de permettre leur rappel à but de réparation. La même choses s'applique au tampon d'édition. Si vous copiez une commande vers le tampon d'édition en entrant une ligne vierge, ou si vous éditez le tampon d'édition, cette valeur du tampon d'édition n'ira jamais dans l'historique des commandes à moins que vous ne l'exécutiez.

Écriture Shell et "tubes" en ligne de commande

(Shell scripting and command-line piping)

You normally use non-interactive mode for input piping. You specify "-" as the SQL file name. See the Piping and shell scripting subsection of the Non-Interactive chapter. 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. 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. Important

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

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

Note

The --sql 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/hsqldb.jar --sql '
      SQL statement number one;
      SQL statement
          number two;
      SQL statement three;
  ' urlid
```

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/hsqldb.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/hsqldb.jar sql... > /tmp/log.sql 2>&1

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

You can also execute SQL files from an interactive session with the "\i"' Special Command, but be aware that the default behavior in an interactive session is to continue upon errors. If the SQL file was written without any concern for error handling, then the file will continue to execute after errors occur. You could run \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 gives error handling statements, they will take precedence.

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

You can use the following SQL file, sample.sql, which resides in the sample directory of your HSQLDB distribution [1]. It contains SQL as well as Special Commands making good use of most of the Special Commands documented below.

/*

```
$Id: sample.sql,v 1.5 2005/05/02 15:07:27 unsaved Exp $
Examplifies 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,
url varchar,
UNIQUE (name)
```

);

\p Creating table pctasklist CREATE TABLE pctasklist (

```
id integer identity,
host varchar(20) not null,
tasksequence int not null,
pctask integer,
assigndate timestamp default current_timestamp,
completedate timestamp,
```

```
  show bit default true,
  FOREIGN KEY (pctask) REFERENCES pctask,
  UNIQUE (host, tasksequence)
```

);

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

\p Inserting test records INSERT INTO pctask (name, description, url) VALUES (

```
  'task one', 'Description for task 1', 'http://cnn.com');
```

INSERT INTO pctasklist (host, tasksequence, pctask) VALUES (

```
  'admc-masq', 101, SELECT id FROM pctask WHERE name = 'task one');
```

commit;

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

```
  java -jar $HSQLDB_HOME/lib/hsqldb.jar  --sql '
      create user tomcat password "x";
  ' mem path/to/hsqldb/sample/sample.sql
```

(The --sql "create...;" arguments create an account which the script uses). You should see error messages betwen 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, and should be used 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 immedately. Piping and shell scripting

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

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

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

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

Example 8.5. Piping input into SqlTool

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

Make sure that you also read the Giving SQL on the Command Line section. The --sql switch is a great facility to use with shell scripts. 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 compatibilty at the cost of poor control and error detection.

```
  *
```

```
    --continueOnErr
```

```
    The output will still contain error messages about everything that SqlTool doesn't like (malformatted commands, SQL command f
  * --autoCommit
```

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

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

Example 8.6. Valid comment example

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

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

Example 8.7. Invalid comment example

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

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

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

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. (But, see the Raw Mode section for an exception). You can, of course, use any SqlTool command at all interactively. I just wanted 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 m
```

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

```
  or if you use \i to read it in interactively, or if another SQL file uses \i to nest it. If \q is encountered, SqlTool will qui
```

```
  \q takes an optional argument, which is an abort message. If you give an abort message, the message is displayed to the user an
```

```
  \q
```

```
  means to make an immediate but graceful exit, whereas
```

```
  \q Message
```

```
  means to abort immediately.
```

\p [text to print]

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

\i /path/to/file.sql

```
Include another SQL file at this location. You can use this to nest SQL files. For database installation scripts I often have a
```

\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.
```

\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
```

\c [true|false]

```
A "true" setting tells SqlTool to Continue when errors are encountered. The current transaction will not be rolled back upon SQ
```

```
With database setup scripts, I usually find it convenient to set "true" before dropping tables (so that things will continue if
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 encounte
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
```

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. Getting Interactive Functionality with SQL Files

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

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

or

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

Character Encoding

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

java -Dsqlfile.charset=UTF-8 -jar $HSQLDB_HOME/lib/hsqldb.jar urlid file.sql...

You can also set this per urlid in the SqlTool configuration file. See the RC File Authentication Setup section about that. Generating Text or HTML Reports

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

Procedure 8.4. Writing query output to an external file

1.

2. By default, everthing will be done in plain text. If you want your report to be in HTML format, then give the special command

3. Run the command \o path/to/reportfile.txt. From this point on, output from your queries will be appended to the specified fil

4. 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 turned on HTML mode with \H before, you can run \H again to turn it back off, if you wish.

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

Kinds of output that get teed to \o files

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

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 ":;" to repeat the last SQL command. If I have several complex queries to run, I turn \o off and repeat until I'm finished. (Every time you turn \o on, it will append to the file, just like we need).

Usually it doesn't come to mind that I need a wider screen until a query produces lines that are too long. In this case, stretch your window and repeat the last command with the ":;" Edit Buffer Command. 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 everything they need to know by looking at (and running!) the sample script sample/pl.sql in your HSQLDB distribution [1] and using the *? command from within an interactive SqlTool session as a reference. (By significant shell scripting, I mean to the extent of using variables, for loops, etc.).

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

All other PL commands imply the "*" command, so you only need to use the "*" statement if your

script uses PL variables and it is possible that no variables may be set before-hand (and no PL commands have been run previously). In this case, without "*", your script would silently use a literal value like "*{x}" instead of trying to expand it. With a preceding "*" command, PL will notice that the variable x has not been set and will generate an error. (If x had been set here will be no issue because setting a variable automatically turns on PL variable expansion).

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. Variables

```
  * Use the * list command to list some or all variables; or * listvalues to also see the values.
  * You can set variables using the * VARNAME = value command. This document explains elsewhere how you can set a values to the c
  * You can also set variables using the --setvar command-line switch. I give a very brief but useful example of this below.
  * Variables are always expanded in SQL, Special, and PL commands if they are written like *{VARNAME} (assuming that a PL comman
  *
```

```
    A variable written like /VARNAME is expanded if it begins an SQL Statement. This usage is called PL Aliasing. See the PL Alia
  * Variables are normally written like *VARNAME in logical expressions to prevent them from being evaluated too early. See below
  *
```

```
    You can't do math with expression variables, but you can get functionality like the traditional for (i = 0; i < x; i++) by ap
```

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

```
    i will be a growing line of dots.
```

```
  *
```

```
    Variable names must not contain white space, or the characters "}" or "=".
```

PL Aliases

PL Aliasing just means the use of a PL variable as the first thing in an SQL statement, with the shortcut notation /VARNAME.

/VARNAME must be followed by whitespace or terminate the Statement, in order for SqlFile to tell where the variable name ends. Note

Note that PL aliases are a very different thing from SQL aliases or HSQLDB aliases, which are features of databases, not SqlFile.

If the value of a variable is an entire SQL command, you generally do not want to include the terminating ";" in the value. There is an example of this above.

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

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

(Note that the * here is not the special command "*", but is the special command "\p" because "*{qq}" resolves to "p").

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

Example 8.8. Simple SQL file using PL

```
  /*
```

```
    grantwrite.sql
```

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

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

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

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

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

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

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

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

Logical Operators

TOKEN

```
The token may be a literal, a *{VARNAME} which is expanded early, or a *VARNAME which is expanded late. (You usually do not wan
```

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.
```

- VARNAMEs 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 use SqlTool interactively to verify that SqlTool evaluates logical expressions as you expect. Just run * if commands that print something (i.e. \p) if the test expression is true. Flow Control

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

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

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

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

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

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

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

Below is an example SQL File that shows how to use most PL features. If you have a question about how to use a particular PL feature, check this example before asking for help. This file resides in the sample directory with the name pl.sql [1]. Definitely give it a run, like

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

Example 8.9. SQL File showing use of most PL features

/*

```
$Id: pl.sql,v 1.4 2005/05/02 15:07:26 unsaved Exp $
SQL File to illustrate the use of SqlTool PL features.
Invoke like
    java -jar .../hsqldb.jar .../pl.sql mem
                                                -- blaine
```

- /

- if (! *MYTABLE)

```
\p MYTABLE variable not set!
```

```
  /* You could use \q to Quit SqlTool, but it's often better to just
     break out of the current SQL file.
     If people invoke your script from SqlTool interactively (with
     \i yourscriptname.sql) any \q will kill their SqlTool session. */
  \p Use arguments "--setvar MYTABLE=mytablename" for SqlTool
  * break
```

- end if

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

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

/* This sets the PL variable 'retval' to the return status of the following

```
 SQL command */
```

- retval ~

CREATE TABLE *{MYTABLE} (

```
  i int,
  s varchar
```

); \p CREATE status is *{retval} \p

/* Validate our return status. In logical expressions, unset variables like

```
 *unsetvar are equivalent to empty string, which is not equal to 0
 (though both do evaluate to false on their own, i.e. (*retval) is false
 and (0) is false */
```

- if (*retval != 0)

```
  \p Our CREATE TABLE command failed.
  * break
```

- end if

/* Default Continue-on-error behavior is what you usually want */ \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 int our new table (you should see 3 row update messages)

- foreach VALUE (12 22 24 15)

```
  * if (*VALUE > 23)
      \p Skipping *{VALUE} because it is greater than 23
      * continue
      \p YOU WILL NEVER SEE THIS LINE, because we just 'continued'.
  * end if
  INSERT INTO *{MYTABLE} VALUES (*{VALUE}, 'String of *{VALUE}');
```

- end foreach

\p

- themax ~

```
/* Can put Special Commands and comments between "* VARNAME ~" and the target
  SQL statement. */
```

\p We're saving the max value for later. You'll still see query output here: SELECT MAX(i) FROM *{MYTABLE};

```
/* This is usually unnecessary because if the SELECT failed, retval would
  be undefined and the following print statement would make SqlTool exit with
  a failure status */
```

- if (! *themax)

```
  \p Failed to get the max value.
  /* It's possible that the query succeeded but themax is "0".
     You can check for that if you need to. */
  * break
  \p YOU WILL NEVER SEE THIS LINE, because we just 'broke'.
```

- end if

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

\p \p Everything worked.

Chunking

We hereby call the ability to transmit multiple SQL commands to the database in one transmission chunking. Unless you are in Raw mode, SqlTool only transmits commands to the database engine when it reads in a ";" at the end of a line of an SQL command. Therefore, you normally want to end each and every SQL command with ";" at the end of a line. This is because the database can only send one status reply to each JDBC transmission. So, while you could run

```
  SELECT * FROM t1; SELECT * FROM t2;
```

SqlTool can only display the results from the last query. This is a limitation of the client/server nature of JDBC, and applies to any JDBC client. There are, however, situations where you don't need immediate feedback from every SQL command. For example,

Example 8.10. Single-line chunking example

```
  INSERT INTO t1 VALUES(0); SELECT * FROM t1;
```

It's useful because the output of the second SQL command will tell you whether the first SQL command succeeded. So, you won't miss the status output from the first command.

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 general reason to chunk SQL commands is if your database requires you to send multiple commands in one transmission. This is often the case when you need to tell the database the SQL or PL/SQL commands that comprise a stored procedure, function, trigger, etc. How?

The most simple way is enter as many SQL commands as you want, but just do not end a line with ";" until you want the chunk to transmit.

Example 8.11. Multi-line chunking example

```
  INSERT INTO t1 VALUES (1)
  ; INSERT INTO t1 VALUES (2)
  ; SELECT * FROM t1;
```

If you list your command history with \s, you will see that all 3 SQL commands in 3 lines are in one SqlTool command. You can recall this SqlTool command from history to re-execute all three SQL commands.

The other method is by using 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.

This paragraph applies only to interactive usage. Interactive users may may end the raw input with ":." instead of ".;". This will just save the input to the edit buffer so that you can edit it and send it to the database manually. You can look at the edit buffer with the ":b" Buffer Command. You would normally use the command ":;" to send the buffer to the database after you are satisfied with it. You'll notice that your prompt will be the continuation prompt between entering "\." and terminating the raw input with ".;" or ":.".

Example 8.12. Interactive Raw Mode example

```
  sql> \.
  Enter RAW SQL.  No \, :, * commands.
  End with a line containing only ".;" to send to database,
  or ":." to store to edit buffer for editing or saving.
  ------------------------------------------------------------
  raw> line one;
    +> line two;
    +> line three;
    +> :.
  Raw SQL 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 in statement [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. PL/SQL 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 processed on the server side and you can only use it of your database supports it. You can not intermix PL and PL/SQL (except for setting a PL variable to the output of PL/SQL execution), because when you enter PL/SQL to SqlTool that input is not processed by SqlFile.

Use Raw Mode to send PL/SQL 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" or "BEGIN", 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 HSQLDB distribution [1]. This script will only work if your database engine supports standard PL/SQL, if you have permission to create the table "T1" in the default schema, and if that object does not already exist.

Example 8.13. PL/SQL Example

```
/*
* $Id: plsql.sql,v 1.4 2007/08/09 03:22:21 unsaved Exp $
*
* 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 you want: Whatever you enter--blank lines, comments, everything-- will be transmitted to the database engine.

Using hsqltool.jar and hsqldbutil.jar

This section is for those users who want to use SqlTool but without the overhead of hsqldb.jar (or who want to use a new SqlTool build with an older HSQLDB distribution).

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

hsqltool.jar includes the HSQLDB JDBC driver. If you do not need to connect to HSQLDB databases at all, then hsqldbutil.jar is what you want. hsqldbutil.jar contains everything you need to run SqlTool and DatabaseManagerSwing against non-HSQLDB databases... well, besides the JDBC drivers for the target databases.

The HSQLDB distribution doesn't "come with" pre-built hsqltool.jar and hsqldbutil.jar files. You need to "build" them, but that is very easy to do.

These instructions assume that you are capable of running an Ant build. See the Building HSQLDB chapter if you need more details than what you see here.

```
1.

   Download and extract a current HSQLDB distribution. If you don't want to use the source code, documentation, etc., you can us
2.

   Cd to the build directory under the root directory where you extracted the distribution to.
3.

   Run ant hsqldbutil or ant hsqltool according to the criteria above. (If your goal is to use this jar with an older HSQLDB dis
4.

   If you're going to clean up afterwards, copy the jar that you built out of lib to a safe location first.
```

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

If you are using a non-HSQLDB JDBC driver, you must set your CLASSPATH to include this new jar file and your JDBC driver, then run SqlTool like

```
     java org.hsqldb.util.SqlTool ...
```

You can specify your JDBC driver class either with the --driver switch to SqlTool, or in your RC file stanza (the last method is usually more convenient).

Delimiter-Separated-Value Imports and Exports Note

This feature is independent of HSQLDB Text Tables, a server-side feature of HSQLDB. 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.

This feature is what most business people call "CSV", but these files are more accurately called Delimiter Separated Value files because the delimiter is usually not a comma, and, more importantly, we purposefully choose an effective delimiter instead of the CSV method of using a delimiter which works in some cases and then use quotes and back-slashes to escape occurrence of the delimiter in the actual data. Just by choosing a delimiter which never needs escaping, we eliminate the whole mess, and the data in our files always looks just like the corresponding data in the database. To make this CSV / Delimiter-separated-value dintinction 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 HSQLDB.

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.

The row and column delimiters may be any String, not just a single character. And just as the delimiter capability is more general than traditional CSV delimiters, the export function is also 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. A benefit to this approach 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.). One 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).

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, HSQLDB identity sequences, etc. Another reason would be to skip binary columns. 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 HSQLDB 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 8.14. 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 characterstics 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 NULLs is [null]. See the next section for how to change these from their default values.

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 8.15. 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.

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/hsqldb.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 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. When specifying delimiters, you can use the escape seqpences \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
```

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 8.16. DSV Export of an Arbitrary SELECT Statement

```
  * *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.

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.

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 require quoting. 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, and the comment character can be changed from its default value.

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.

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 multple 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 8.17. 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 8.18. 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. Unit Testing SqlTool

The SqlTool unit tests reside at testrun/sqltool in the HSQLDB source code repository. Just run the runtests.bash script from that directory to execute all of the tests. Read the file README.txt to find out all about it, including everything you'd need to know to test your own scripts or to add more unit test scripts for SqlTool.

[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 HSQLDB 1.9.x distributions, namely, HSQLDB_HOME/sample. HSQLDB 1.8.x users should translate these sample directory paths to use HSQLDB_HOME/src /org/hsqldb/sample/....