TdBench 8.01 User Guide

A TOOL FOR SIMULATING DATABASE WORKLOADS AND CONDUCTING PRODUCTIVE DATA WAREHOUSE BENCHMARKS

teradata.

Table of Contents

Why Do You Need TdBench?

TdBench is a tool specifically designed to simulate database workloads. With this tool, you can:

- Measure performance before vs after a change to add indexes, partitioning, compression, etc
- Measure the impact to your DBMS of changes to settings, a patch, or a new software release
- Simulate a workload for a new application or a proof of concept
- Compare the performance of one platform to another
- Compare performance of different data base vendor's products

TdBench 8.01 works with any database supporting JDBC, can run other command line tools, has a robust scripting language, captures results in an internal database and can interface to DBMS query logging.

Perhaps you have some questions:

Does this just run on Teradata? No, it works with any database supporting JDBC

Where does TdBench run? On Linux or Windows or possibly other platforms that support Java. The first releases used Windows/DOS batch files. Release 5 was posted on the web and presented at Teradata Partners in 2010. Release 7.2 was posted on the web in 2016 and ran on Linux. Release 8.0 was re-written in Java and posted on the web in 2018. Usually, it runs on a client PC/Server. However, it can run on a database node, but it may have a tiny impact on the DBMS you are trying to measure.

The German Nobel-laureate physicist Werner Karl Heisenberg (1901-76) observed:

> "The very act of observing alters the particle being observed and makes it impossible (even in theory) to accurately predict its behavior." … (The Heisenberg principle)

Running TdBench on a node to measure performance of the system including that node running TdBench is a prime example of that impact since TdBench will steal cycles from the node that should be processing the target workload. However, performance of the client, network bandwidth, and latency are also major impacts in the measurement of queries-per-second or response time of tactical queries. With TdBench 8.01, the LATENCY command will issue as many select of a literal in 5 seconds as it can to help you understand the measurement limitations of your configuration.

Is TdBench Free? Yes, however you can't modify it, relabel it, re-sell it … the normal license restrictions you'll find in the setup/license.txt and via "HELP LICENSE" command.

Why did Teradata create TdBench? Originally to improve our productivity, capture benchmark artifacts in a systematic way and to improve analysis of results. Then a DBA at a customer site told us how valuable it would be to use in testing release changes, so we made it public.

Why did Teradata change TdBench to work with competitor's databases? Customers and prospects testing DBMSs from multiple vendors want a single tool for consistent execution. If we want to avoid less productive, lower quality benchmark executions, we need the tool to be usable on other platforms. While we obviously believe in our database product, if the benchmark is fair and another product has features that are a better fit for a prospect's particular needs, we can live with that.

Is this a supported Teradata Product? No. Bugs will be fixed on a best effort of the author. Post ideas or comments on https://github.com/Teradata/tdbench/discussions. If you encounter a bug, the Windows and Linux scripts will create a tdbench_exception.txt file with information that may help in answering what went wrong. You can post details of your problem at https://github.com/Teradata/tdbench/issues.

Interactive or Batch?

Yes!

If you start out using TdBench interactively, you will have access to a coach (the question mark) and help information on the commands. At any time, you can save the commands from a prior test and edit them into a command file that you want to run again and again.

Interactive mode

A shell script is provided for Linux and a batch command file for Windows to validate the environment, run the program and collect information in the event of a problem.

- Linux: ./tdbench.sh
- Windows: tdbench.bat

On first execution, you’ll be asked if you agree to the license and if you respond with “yes” the program will start. Then you will want to use the SETUP command to get information and configure your DBMS.

What Does It Do?

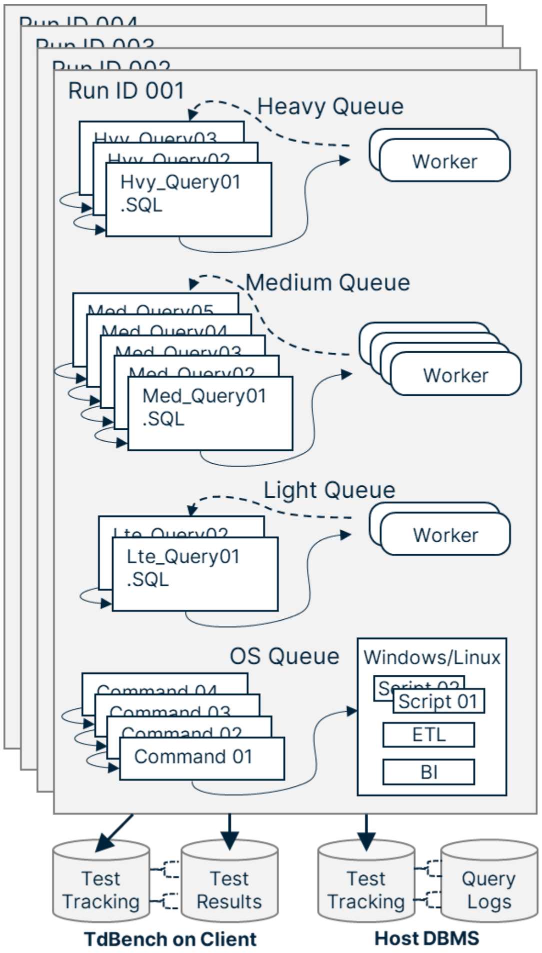
TdBench simulates realistic production systems by allowing definition of the different types of work and adjusting the number of concurrent executions for each type of work.

- It captures the results each query execution in its internal database.
- It facilitates analysis host DBMS resource consumption by maintaining test metadata on the host DBMS to join with its query logs.

Tests are defined with:

- queues of SQL queries and scripts or OS commands
- variable number of execution threads (workers) per queue
- commands to pace queries by time or percentage
- parameterized queries to simulate different users
- optional query prepare reducing DBMS parsing
- scheduled start of processes or individual queries
- Fixed work or fixed period execution models
- Scripting language to automate multiple tests

Tests can be defined as simply as 4 statements. Analysis capabilities have been used to track individual query performance over hundreds of runs during projects with constraints like: WHERE RunID in (79, 81, 105).



Configuring TdBench for your DBMS

TdBench prompt will show the current directory. You should issue the SETUP command to select a DBMS and get help for configuring the startup files OS shell scripts, TdBench startup scripts, and JDBC.

Your First Commands

Issue the command “help firsttime” help providing a brief introduction page to the tool … *since most of us never read manuals like this*. You can use the HELP command by itself or follow with the name of a command, a topic or the word index to get a list of commands and topics. At any point issue a ? by itself and you’ll get hints on what to do next. A simple test which would work for any ANSI standard DBMS , (once you’ve set up a database alias, in this case we will call it “mydb”) is:

```
Define firsttest A test to validate database connectivity
queue q1 select current_timestamp;
worker q1 mydb
run
```

Saving Commands from a Prior Test For Batch Use

Every test is assigned a RunID and the commands and results are saved in an H2 database that is internal to TdBench. Assuming the above test was assigned RunID 1 (after all it was named “firsttest”), you could save the commands for later use with:

- Linux: `save 1 scripts/firsttest.tdb`
- Windows: `save 1 scripts\firsttest.tdb` …. *However save 1 scripts/firsttest.tdb also works*

If there were errors as you typed commands, the error message will be written out with a comment marker (#) after the statement in error. If you successfully ran the test, your corrected command should be later. In any case, edit the file to review what you saved before using in batch mode.

(Hint: all input is saved in the trace buffer. Use “trace clear” at the end of tdbench.tdb to clear the setup statement so they don’t end up as the first rows saved in the script from a prior test)

Batch mode

To run one or more scripts, you can place them on the command line

- Linux: `./tdbench.sh scripts/firsttest.tdb`
- Windows: `tdbench.bat scripts\firsttest.tdb`

You can put multiple tests in one file. It may be more productive to put them in separate files, so you can run one or run them all. You can also use wild card searches in the Linux command line. Example:

```
./tdbench.sh  scripts/serial.tdb  scripts/wkld05.tdb  scripts/wkld10.tdb   scripts/wkld20.tdb
```

You can also put TdBench commands on the command line. If the TdBench script or command placed on the command line has multiple parameters, enclose the entire file/command + parameters in “quotes”. You can also use the -d option to specify which DBMS to use, overriding the setting in tdbench_config.?. For more information, issue:

```
./tdbench.sh  -h
```

Meet Your Coach. The Question Mark

It is a challenge when you are faced with a new tool with so many available commands and scripting options. Unlike us Benchmark Analysts, most ***normal*** people won't use this type of tool daily.

TdBench has always had help built in and as of the writing of this document there are 69 help files built into TdBench for topics and commands. If you issue:

If you issue	You'll get
Help	A brief summary of the main test definition commands and a list of topics
Help ?	Syntax for help plus a listing of all help file names
Help index	A listing of help grouped by command type, topic and control files

For any command, if you issue that command followed by a question mark, you'll get a brief syntax of that command. For example:

```
worker ?
WORKER command syntax: [queue-name] db-alias-name {count} | OS {count}
```

The square brackets [ ] mean it is required, the curly brackets { } mean it is optional, and the vertical bar indicates one or the other.

But that doesn't help if you don't know where to start. That is why the coaching facility was added to TdBench 8.xx.

The following is a sample of the type of information the Question Mark Coach will provide:

1st session	? There are no database aliases defined yet. Use CLASS and DB statements to define them. For more information, use HELP CLASS and HELP DB
After defining aliases	? You have the following database aliases defined: sb00 user:de110922 on jdbc:teradata://10.25.11.107 vm user:acme_benchmark on jdbc:teradata://192.168.202.128 To start a test, use the DEFINE statement. For more information, use HELP DEFINE.
After define	? You have defined a test, but you haven't defined any queue(s) or their worker(s). For more information, use HELP QUEUE and HELP WORKER
After queue	? So far, your test is defined as: TestName:firsttime Description:A test to validate database connectivity #Queues:1 Queue:q1 #Cmds:1 #Params:0 #Workers:0 You have defined 1 queue(s) but one or more is missing workers. For more information, use HELP WORKER

After Worker	? So far, your test is defined as: TestName:firsttime Description:A test to validate database connectivity #Queues:1 Queue:q1 #Cmds:1 #Params:0 #Workers:1 Worker 1: de110922 connection:jdbc:teradata://10.25.11.107 If you want information on one of the queues, use ? queue-name. Next step is to use the run statement to execute firsttime. You can put a time specification like 5m for 5 Minutes after RUN. For more information, use HELP RUN
Asking about a queue	? q1 Queue:q1 #Cmds:1 #Params:0 #Workers:1 Statements: 1) select current_timestamp;
After a run	? Your last test named: firsttime started: 2018-09-29 08:16:56.235 and ran for 0 seconds with 1 statement and 0 errors. You may start another test with DEFINE or issue QUIT to exit

Back to the question posed on the last section of this manual:

Interactive or Batch?

A better answer than “yes” is**:**

Interactive first, then batch later.

The One Page TdBench 8.0 User Guide

Before starting, you need to load the JDBC drivers for your database with the CLASS command and define a database alias including username and password for that class with the DB command. Either user the HELP CLASS and HELP DB commands for more information or see One Tool, Many Databases.

An example script:

```
define serial Execute all queries one at a time
queue queries scripts/queries/*.sql
queue queries call acme_benchmark.monthend_rpt('2017-11-30')
queue queries os scripts/etl_monthend.sh
worker queries mydb
run
```

The first word after the **DEFINE** statement is the short test name and is followed by a descriptive title. A RunID is sequentially assigned to each test and is used to organize the test artifacts.

The **QUEUE** command loads the queries from the specified directory into a queue named “queries”. You may have multiple QUEUE statements to the same queue name to collect queries from different directories or with specified naming patterns. You may have multiple queues defined, each with their own set of queries. New for TdBench 8.01, if the first token on the string entered on the QUEUE is not a file name, it is taken as an SQL statement. (A semicolon is no longer needed). If the first word after the queue name is “OS”, that command or shell script will be executed at that point in the queue.

Every queue must have at least one **WORKER** associated with it. If that worker is to execute SQL commands, it must have a database alias specified which in the example above is “mydb”. By default, that will create one worker. If you specify a count after the database alias, then that number of workers will be started with each one taking the next query off the queue to execute as the test starts or as the worker completes a query. Instead of a database alias (e.g. “mydb”) you can specify “OS” if the queue has only OS commands and no SQL statements.

The RUN statement initiates the processing of the queued work by the workers. If specified without any parameters, the work in the queues will be executed once and the test terminated. TdBench supports:

- Fixed work - The queued work is executed one time with: run
- Fixed period initiation – work in the queue is initiated/cycled until the period elapses: run 30m
- Fixed period execution – queries in flight are terminated at end of test: run 30m kill
- Mixed – the run statement specifies duration, but STOP is put into a queue (e.g. etl only once)
- Master/noise – the QUEUE statement is used to put KILL ALL or STOP ALL on one queue

The time specification after the run statement can be in seconds, minutes or hours. The following are equivalent: 3600s = 60m = 1h or 30s = .5m.

After the test completes you can get results with the following commands:

Select * from testtracking where runid = *nn*;	Puts out 1 line summarizing RunID *nn*
Select * from testresults where runid = *nn*;	Puts out 1 line per query executed in RunID *nn*
List -5	Lists out the last 5 tests. Without a number, it lists all

Quick Command Reference:

Some of these commands are discussed in this User Guide. For information on the others, use the Help command under TdBench. The commands are not case sensitive. Linux file references are case sensitive.

Primary TdBench Commands

# comment
Documents scripts. (Blank lines are ignored.)

?
Request current in-progress setup of a test and get suggestions of what else is needed

define [test-name] {test description}
Define name and description of a test.

help [topic | command | ?]
Get details on the syntax and usage of a command

queue [queue-name] [query; | "OS" command; | pathname | wildcard pathname | "kill | stop" {"all"};}
Specify a queue of commands or queries/subqueries for the test.

worker [queue-name] db-alias-name {count} | “OS” {count}
Defines workers to run the queue commands.

run { # [“h”|”m”|”s”] {kill}.}
Starts the test.

quit [return code] or exit [return code]
Exits TdBench setting a return code that can be tested by an OS script.

Environment/Script Control Commands

cd [directory]
Change TDBench working directory.

class [class-name-of-JDBC-driver] [JDBC-protocol e.g. "jdbc:teradata"] [file-name(s)-of-JDBC-driver-jar]
Load JDBC driver class from a file.

db [alias-name] [JDBC-protocol e.g. “jdbc:teradata://192.168.1.28/database=benchmark”] {username {password}}
Define a database alias for use in WORKER and/or SQL commands.

echo {text:filename | append:filename} {comments + variables} {delim=*string_on_line_later*}
Displays text on console or writes text (with variables) to a file. Can be multi-line with delim=

exec [filename|pathname] {argument-1 ...}
Alias for include. Read TDBench commands from a file (can be nested and allows parameters :i1, :i2 …)

include [filename|pathname] {argument-1 ...}
Read TDBench commands from a file (can be nested and allows parameters :i1, :i2 …)

Latency [alias-name]
Measures how many queries can be executed in 5 seconds and its impact to short running queries

sleep { # {“s" | "m" | "h"} } | {yyyy-mm-dd-}hh:mm{:ss}
Will stop the executing the script until a time or a relative time has passed.

status [# seconds | 0]

Will set the frequency of status messages during the test execution. Default 20s

trace [on | off | list | save filename | clear]
Controls echo of statements to console, or lists out inputs and outputs or saves them to file

zip [work-directory-name] [zip-directory-name] {erase}
Defines a directory to archive using zip protocol after test.

Commands to Insert Commands During Test

BEFORE_RUN command syntax: before_run ["SQL" ... sql command; | "OS" ...os command; | "DELETE"]
Statements to be run before RUN statement. (allows substitution of TdBench variables)

AFTER_RUN command syntax: after_run ["SQL" ... sql command; | "OS" ...os command; | "DELETE"]
Statements to be run after the test completes.

BEFORE_WORKER command syntax: before_worker [queue-name] [query; | "OS" command; | pathname]
Specify one or more queries to run in each worker before a queue. (e.g. database command)

BEFORE_QUERY command syntax: before_query [queue-name] [query;]
Specify a query or command to run before each subquery in a queue. (e.g. Query_Band)

AFTER_QUERY command syntax: after_query [queue-name] [query;]
Specify a query or command to run after each subquery in a queue.

AFTER_SQL command syntax: after_sql [tdbench-command | "delete"]
Specify command(s) to execute to evaluate a SQL command :retcode or :linecount

AFTER_OS command syntax: after_os [tdbench-command | "delete"]
Specify command(s) to execute to evaluate an OS command :retcode

Test Control Commands

at [queue-name] [#]{"s" | "m" | "h"}
Time or relative time to start a queue.

explain [queue-name]
Adds "explain" to every subquery in a queue.

finish [queue-name]
Ensures queue must finish at least once before the test ends.

limit [queue-name] [#]{"s" | "m" | "h"}
Maximum time to run a subquery in a queue before aborting.

pace [queue-name] [#]{"s" | "m" | "h"}
interval between queries in a queue.

param [queue-name] [file delimited parameters] {delimiter | tab}
Provides parameters for subqueries in a queue.

prepare [queue-name] {data-type-1, ...}
Provides parameters for subqueries in a queue run using prepared statements.

rowcount [queue-name]
Workers in a queue will count all returned rows (affects performance!)

rows [queue-name] {maximum row count}
Workers in a queue will returns some or all rows (can affect performance!)

Queue Preparation Commands

replicate [queue-name] [multiplier]

Duplicates a queue's commands by a multiplicative factor.

save [queue-name | runid] [pathname-template-to-save-into]

Saves a queue's commands in a set of files for future tests or the statement in a runid into a file.

shuffle [queue-name]

Reorders a queue's commands and parameters randomly.

Immediate Commands

goto [labelname | testname]

Skips until LABEL or DEFINE statement with the label/test name specified

if [variable] [= | != | > | >= | < | <=] [constant] ["THEN" statement]

Tests results of prior OS, SQL or RUNs to determine next statement

label [labelname]

Provides a name to be referenced by GOTO

OS { file:fileame | file:null } [command-to-execute]

Immediately executes an operating system command

sql { file:filename | file:null | text:filename} [db-alias-name] [sql-command";" | filename-of-sql-commands]

Immediately executes one or more SQL statement

H2 (Results) Database Commands

archive [tables-prefix-name] {"from" other_prefix} {"where" where-conditions]

Archives tests in H2 in separate tables. Optionally may copy from another archive and apply where…

delete [runid]

Removes rows from TestResults table. Useful when test fails with thousands of errors

Dump {archive-name} {"to" zip-prefix}

Writes all test information from H2 archive to CSV files and zips as archive-name.zip or zip-prefix.zip.

list {archive-name} {-count | -recent-count | runid {to runid] | "where" where-conditions}

Lists all tests captured in the H2 database.

note [runid] {add} {notes or observations to set, add, or omit to delete note}

Adds a note to a RunID describing conditions or usefulness of test to final results

remove [tables-prefix-name]

Deletes archives data in H2

Restore [archive-prefix] {"to" other-archive-prefix}

Restores tests archived/dumped from a zip file to tables with archive-prefix or other-archive-prefix

select [h2 SQL select query]

Allows query against test results output to console (alias for SQL H2 SELECT...).

H2 Reporting Views:

rpt_tests – Summarizes at the RunID level.
rpt_queues – Summarizes by queue within each RunID.
rpt_querys – Summarizes by query name within queue within each RunID.
rpt_errors – Summarizes errors by RunID.

One Tool, Many Databases

TdBench 8.01 may be downloaded from https://downloads.teradata.com. You will want to make sure to select TdBench 8.01,xx. (TdBench 7.2 and TdBench 5.0 may still be online). It is downloaded as a zip file. While you are on that site, if you are going to be using Teradata, download the JDBC driver if you don't already have it installed on the PC/server where you will be running TdBench. It will be easiest if it is in the same directory where you place TdBench.

It unzips into a directory with the tdbench.bat and tdbench.sh scripts. These are needed to validate the environment, start the program and on exit, if there are errors, collect information to enable us to diagnose problems with the application.

There may be updated support for other DBMSs at https://github.com/Teradata/tdbench. Compare the version_history.txt file on the setup directory for the DBMS versus the version_history.txt file on GitHub.

Getting JDBC Drivers for Your Database

The basic process is to search the web for your vendor's JDBC driver, e.g. "Redshift jdbc driver download" and place that file in the same directory where you unzip TdBench 8.01. (That directory will have the files tdbench.sh, tdbench.bat, etc).

On your first usage of TdBench, you will want to use the SETUP command to create the startup files that run for every TdBench session to load the JDBC driver and use it to define connection(s) to your DBMS.

Define Once, Use Many – tdbench.tdb

Rather than having to put this logon information in many scripts or having to remember how to enter it each time, you can put the CLASS statement to load your JDBC driver and DB statement to define the logon information into the tdbench.tdb file that is loaded every time TdBench starts.

If you don't want to put your password into the tdbench.tdb file, you can define an environment variable to hold your password and either prompt for it each time TdBench is started. To set the password:

Linux:

```
export mypw=Xpassword123      … or
read -p "Enter password for DBMS logon: " mypw
```

Windows:

```
set mypw=Xpassword123                    (Windows) … or
set /p ans="Enter password for DBMS logon: "
```

then an example of your DB alias for either OS in would be:

```
db sb00 jdbc:teradata://10.25.11.107 MyUser ${mypw}
```

In addition to the CLASS and DB statements, you may also want to put in the following statements in your tdbench.tdb file:

BEFORE_RUN	Statements to be executed immediately before the RUN statement is processed. For Teradata, we use this to execute xxx_benchmark.teststart( ) which creates a row in a testtracking table with the test name, information collected about the current DBMS environment, and importantly the precise timestamp when the test started. For other DBMSs, this can be a simple insert statement of the starting timestamp along with the TdBench variables :runid, :testname, :testdescription, and :runsecs.
AFTER_RUN	Statements to be executed after the test is complete. For Teradata, we use that to execute xxx_benchmark.teststop( ) which updates the row in the TestTracking table for the RunID with the precise timestamp when the test ends plus ending DBMS environmental information. The starttime and actualstoptime in the TestTracking table are joined in reporting views to various Database Query Logging and Resource Usage tables. For other DBMSs, this can be a simple update statement with the ending timestamp along with the TdBench variables :resultcount and :errorcount.
AFTER_NOTE	This SQL statement is executed after the NOTE command is used to update the Notes column in the internal H2 TestTracking table. It can be used to post the same note to the host DBMS TestTracking table using the TdBench variable :note for the :runid.
ZIP	Defines a directory where you want to output logs from other programs you run during a test and a directory where you want those to be zipped and associated with a RunID.
STATUS	Specify the monitoring frequency of tests to the console log in seconds, minutes or hours (s, m h). By default, status will output every 20 seconds. For status every 2.5 minutes, use: status 2.5m Status 0 will cause one line to be written to the console as every query completes execution with much of the information that is in the TestTracking table.

The Worker Statement

The DB alias is used in the worker statement for logging on to the database for stream(s) of queries. (The DB alias is also used in the SQL command). The alias is followed by a repeat count. By default, it will logon with the logon/password specified in the DB Alias. It is possible to override logon and password. The following are examples:

Worker q1 td 5	5 workers are assigned to q1 with logon/password from DB alias
worker q1 td(datamining)	overrides username in DB alias, same URL & password
worker q1 td(datamining/diffpwd)	overrides the username and password in DB alias
worker q1 td(rpt01:10) 5	says rpt01 - rpt10 available, but only first 5 will be used
worker q1 td(rpt1:10) 5	INVALID - must have same number of digits in root logon name
worker q1 td(rpt10:5) 8	works, but creates rpt10, rpt20, rpt30 … since :5 was 1 digit
worker q1 td(rpt11:15) 20	creates 20 workers with 4 sessions @ for rpt11 - rpt15
worker q1 td(tac01:05/diffpwd) 5	logon 5 users with a different password than the DB alias.

Where to Run TdBench 8.0?

TdBench 8.0 has been written in Java, so it should run on any platform that supports Java, however, we've only tested it with Linux and Windows. Some report using it on Mac. (Provide feedback at https://github.com/Teradata/tdbench/discussions if you are using it on other platforms). However, as TdBench 8.0 interacts with the OS, there are some differences and a few oddities that are documented here to decrease the chance you'll be debugging a script.

For both environments, the TdBench commands and their defined parameters are not case sensitive. QUEUE, Queue and queue are equivalent as are "RUN 1m", "Run 1M KILL" and "run 1m kill".

Windows:

- File names are not case sensitive
- File name references used by TdBench commands can use the forward slash / or back slash \ as a separator between the elements of the path name. TdBench will use the correct one for the OS.
- When running an OS command, the string following OS may contain Windows options which use the forward slash, so TdBench does not process / vs \. It is further complicated since Java takes \ to be an escape character. Therefore, the strings containing commands and file references following the TdBench OS command on Windows must use a double back slash. For example:
  ```
  os type scripts\\sample.tdb
  ```
- If you launch a full screen program like NotePad from the TdBench OS command, the OS command will not complete until you exit that application.
- By default, the variables :edit and :view use NotePad, but you can use the SET statement to use other commands for editing. If directory names include spaces, enclose in quotes. Example:
  ```
  ……set edit = os "C:\\Program Files\\Notepad++\\notepad++.exe"
  :edit scripts\\sample.tdb
  ```

Linux:

- File names are case sensitive
- File name references use the forward slash / in all cases
- New for TdBench 8.01, if you launch the full screen Linux programs vi or less, it will work but other full screen applications may cause TdBench to hang.
- By default, the variable :edit is set to os vi and :view is set to os less, but you can use the SET statement to use other commands for editing and viewing. Example:
  ```
  :edit scripts/sample.tdb
  ```

The last consideration is the type of platform. Normally, TdBench runs on a client Server or PC (like your laptop). For a benchmark with many worker sessions, you should use a client with multiple CPU's. If you are going to have a high volume of very short running queries and you are trying to measure **Q**ueries **P**er **H**our (QPH), you would want the client to have low latency to the DBMS. On one benchmark, we started running TdBench on a laptop connected to VPN connected via Axedia to a customer DBMS. Running a trivial statement that didn't actually retrieve data executed 300 statements in 30 seconds from the laptop and when we moved to running on a server in their data center, we executed 3,000 statements in 30 seconds.

New for TdBench 8.01, we've added the LATENCY command that will issue trivial queries for 5 seconds to show you the impact of latency with your selected client. Example of output from 2 platforms: a Linux server in the same data center as the DBMS and a Windows laptop running via VPN from Ohio to San Diego:

```
Starting 5 second test ...There were 6147 executions in 5 seconds for average 0.0008 seconds@
Starting 5 second test ...There were 12 executions in 5 seconds for average 0.4167 seconds@
```

There are TdBench commands that force the application to retrieve rows. Those are the ROWS command where you specify how many rows should be returned to the console and the ROWCOUNT where TdBench will read and count all result rows. This performance is highly influenced by things outside of the DBMS. **The purpose of TdBench is for benchmarking the DBMS**. The transfer time thru the network and the client processing of the rows may hide the differences in DBMS performance. If the query has a GROUP BY or ORDER BY clause, the DBMS must complete the processing of the query before returning the first row. By default, TdBench will execute the query and create the result set but will not retrieve the rows. It may be justified to use the TdBench ROWCOUNT or ROWS command for one or two serial tests to validate that all DBMSs are producing equivalent results, but not for all tests.

Basic Scripting

It is highly recommended that your first scripts be composed interactively so you have access to help facilities and the question mark to get feedback about what you've defined so far. You can then use the SAVE command on a completed test execution to save those commands created interactively into a .tdb file. (The .tdb suffix isn't required. It can be anything you want). Hint: Use the "TRACE CLEAR" command at the end of tdbench.tdb to prevent saving all the statements that were executed during startup. That will reduce your editing of the output of the SAVE command.

While it is easy to compose a test with complicated logic that is set to run for hours, it is easy to waste hours and submit millions of failing queries from a test that you haven't tested. Frequent culprits are

- Syntax errors in queries,
- missing tables and views, and
- lack of sufficient rights by the logon ID(s) used for the query driver to access the tables

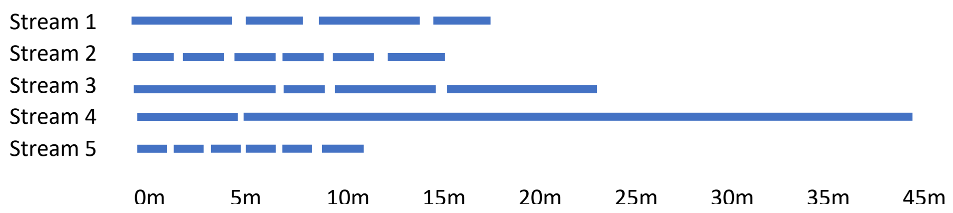
Your first test should always be a serial test of all queries. There are 2 ways to do that:

- Against empty tables before you load data. (*This is the fastest your benchmark will every run* 😊). It is not only productive, but if your DBMS can log object references, you can analyze which tables are required which can save you time exporting/importing data for the test.
- Against fully loaded tables. This allows you to see what percentage of either CPU or I/O is used by queries, allowing you to guess at how many streams of workload will fully saturate your platform. For example, if it looks like 20 streams will fully use the CPU, a good set of workload tests would be 5, 10, 20, 40 and 80 concurrent streams. (See discussion on concurrency)

Fixed Work or Fixed Period?

The serial test is an example of a "Fixed Work" test which means that all the defined work is executed one time in one queue by one worker. This measures the ability of the DBMS platform divide the problem into parallel activities and muster the maximum resources to complete the queries as fast as possible. It is the performance you'd get if you found a time when you were the only user of the platform.

When a "Fixed Work" test is used with multiple workers, unless all the queries are of equal size, the actual concurrency will not be equal to the number of workers. We've observed tests designed by customers and prospects with simple scripts initiated all at once with heavy and light queries and they measure the elapsed time for all work to complete. We refer to this as a "Decay Test" because the concurrency continues to reduce until there is only one query running which defines the duration of the test. Graphically:



A fixed work test is initiated by the RUN command with no duration specified. Example:

```
run
```

A fixed period test will cycle the queued work until the period elapses. There are two variations:

- **Fixed Period Initiation** meaning the queries will be initiated during the specified time. Example:

  ```
  run 30m
  ```

- **Fixed Period Execution**, meaning the queries must complete during the specified time. Example:

  ```
  run 30m kill
  ```

The duration can be specified in seconds, minutes, or hours. The following are equivalent: 3600s, 60m and 1h.

For Teradata, the TestStart macro accepts the “ReportingSeconds” which is the interpretation of the time specification on the RUN statement and is used in the reporting views to set the column “FinishedInTime” to Y or N for the Fixed Period Initiation test, allowing analysis of the queries in-flight at the end of the test and estimating a crude fractional completion.

ETL activities modify tables and often can only be executed one time, requiring them to be a “Fixed Work” test. Within a fixed period test, you can cause one queue to only execute once by putting the word “stop” in the queue after you have put in all the required ETL activities. Last queue statement example:

```
queue etl stop
```

Sometimes you want to focus your testing on the ETL activities but would like other background noise queries to be executing as long as the ETL activities take to run. For that case, you would put the words “stop all” or “kill all” at the end of the queue or the parameter file which would stop the test. The special commands (not case sensitive) you can put in the queue are:

stop	Prevents any further submission of queries from any queue
stop.me	Prevents any further submission of queries from this queue
stop.*queuename*	Stops further submission of queries a given queue name
stop.all	Stops further submission of queries in any queue. Stops subsequent tests
kill	Kills all queries in flight for all queues, stopping the current test.
kill.*queuename*	Kills queries in flight in a given queue name
kill.all	Kills all queries in flight for all queues, and stops this and subsequent tests.

One use of the above stop/kill statements is when using the query driver with a single template query and a parameter file that contains a single query such as an airline reservation query and the parameter file contains a list of PNR keys (Passenger Name Record keys) that must be looked up. By specifying a long run time (e.g run 8h) and putting the word “stop” on the last row of the PARAM file associated with the queue, the specified reservation retrievals will be executed by as many workers as you specify. We’ve also used this for mass loading lots of tables with an OS command in the queue that takes a file name and table name as parameters and a PARAM file that has the mapping of file name to table name.

Even more sophisticated stopping of parts of the test are possible using external flag files that are placed in the home directory of TdBench by an OS command within a queue or by an external process (such as a benchmark analyst that realizes the test is going horribly wrong and needs to be killed). Those (case sensitive) files are:

stop.all	Stops further submission of queries in any queue and subsequent tests
stop.queuename	Stops further submission of queries in the given queue name
kill.all	Kills all queries in flight for all queues. Stops subsequent tests.
kill.queuename	Kills all queries in flight for the given queue name.

Concurrency

Frequently we are told that a company has 500 users so we need to run a test with 500 sessions (using a count on the WORKER statement). It is highly unlikely that all 500 would ever be logged on at the same time. Further, of those that are logged on, it takes time to compose queries, review results, get coffee, take calls, chat with the co-worker over the partition, etc. When the TdBench executes queries, by default, they are executed without any “think time”. We’ve seen some vendor’s query drivers with a parameter for “Think time” which is a ploy to reduce concurrency. They report 100 concurrent users but with 15 second think time on 5 second average execution time queries, there would only be an average of 25 concurrent sessions.

The following are standard industry rules of thumb for a decision support environment:

- If you have 500 users, (sometimes referred to as “Seats”)
- 1/10th or 50 would be logged on a peak period
- 1/10th of logged on users would have queries in flight (or 5 WORKERs).

One way to measure actual concurrency is to join the query logs against a derived table that has a timestamp every minute and count how many queries have a start time and stop time surrounding that timestamp. If you can differentiate different types of users, that will lead you to defining the different queues for your tests and the ratio of workers in each.

One Queue or Many?

Generally, not all users need the same type of response time from queries:

- The call center needs consistent quick response time so they can service customers
- The user using a BI tool with Metadata hosted on the database needs fast response on the quick lookups of dimensions so they can productively build reports
- The person in accounting needs fairly good response time, but may have a slower SLA than the above users
- The data mining analyst is generally processing lots of data with intense processing, so it is natural to expect these queries to have a longer SLA than the above users.

Your analysis of the workload you are trying to simulate should break the run time and the count of queries in flight by user as a starting point for designing the number of queues of work. The ratio of the number of workers per queue should match the typical ratio of the number of queries in flight.

If a type of workload has a wide range of query execution times, it is a good practice to split the long running queries from the short ones to prevent clogging up the workers with all long running queries and killing the Query per Hour metric (QPH). As an exaggerated example: If the queue has 9 queries that run 1 second and 1 query that run 1 seconds, with 5 workers, after several cycles, most workers will be running long queries. Example:

Time	Worker 1	Worker 2	Worker 3	Worker 4	Worker 5	% Heavy
0	1 sec	1 sec	1 sec	1 sec	1 sec	0%
1	1 sec	1 sec	1 sec	1 sec	**10 sec**	20%
2	1 sec	1 sec	1 sec	1 sec	**10 sec – 9 left**	20%
3	1 sec	1 sec	1 sec	1 sec	**10 sec – 8 left**	20%
4	1 sec	**10 sec**	1 sec	1 sec	**10 sec – 7 left**	40%
5	1 sec	**10 sec – 9 left**	1 sec	1 sec	**10 sec – 6 left**	40%
6	1 sec	**10 sec – 8 left**	1 sec	1 sec	**10 sec – 5 left**	40%
7	1 sec	**10 sec – 7 left**	**10 sec**	1 sec	**10 sec – 4 left**	60%
8	1 sec	**10 sec – 6 left**	**10 sec – 9 left**	1 sec	**10 sec – 3 left**	60%

The following is an example of a test with multiple queues, both for different organizations and segregating long from short running queries to maintain a ratio that matches production:

```
define wkld10 Workload with 10 streams for 30 minutes
queue finance scripts/finance/*.sql
worker finance mydb 5
queue salesshort scripts/sales/short/*.sql
worker salesshort mydb 4
queue saleslong scripts/sales/long/*.sql
worker saleslong mydb 1
run 30m
```

We've seen users of prior releases of TdBench setup 10-20 query queues. That was done for various artificial reasons versus the detailed analysis of the actual workload queries in flight for various SLA's. One problem is that there weren't enough queries in any one queue to support a realistic test and in some cases, with more workers than queries in a queue, a few workers were running the same query at the same time. While there probably isn't a limit to the number of queues, there probably isn't a good justification to have more than 5-6.

Timed Events – The AT command

One of the simulations you may want is to start a particular queue of work at some time into a test. You can use the AT command to specify the time relative to the start of a test when a queue should start processing its work. It could be used to:

- Introduce poorly constructed queries at some point in a test to demonstrate how the DBMSs workload management handles the situation
- Introduce ETL into the workload to demonstrate whether the DBMS can maintain query response time
- Ramp up query workload for some period. For example, you could start a queue with multiple workers and lots of fast running queries with a STOP to execute them once to see if the DBMS

can switch focus to giving those short queries good response time or can automatically scale up or scale out the number of nodes processing the workload.

Example:

```
Define mixedwl10s2e Workload with 10 concurrent query streams and 2 etl streams
queue sales scripts/sales/*.sql
worker sales mydb 10
queue etl scripts/etl/*.sh
worker etl os 2
at etl 5m
run 1h
```

Query Replay:

Instead of delaying an entire queue, you can delay any line in queue to execute at a time relative to the start of the test. No matter how many workers are assigned, the query with a time specification will stop the queue until that time relative to the start of the test. This is most appropriate for a fixed work test since in a fixed period test, once the time specified on that queue statement has passed, the query will execute immediately on subsequent cycles of the queue.

In the following example, we will use in-stream queries to demonstrate the replay and cause the output of each query (up to 10 rows) to be displayed on the console.

```
define timed Demonstrate replay of queries
queue q1 at 5s select current_timestamp;
queue q1 at 10s select current_timestamp;
queue q1 at 15s select current_timestamp;
worker q1 mydb
rows q1 10
run
```

The output from the above test to the console was:

```
2018-10-01 14:34:03.859: All worker(s) started at 2018-10-01 14:34:03.858.
Current TimeStamp(6)
1: 2018-10-01 11:34:08.92
Current TimeStamp(6)
1: 2018-10-01 11:34:13.91
Current TimeStamp(6)
1: 2018-10-01 11:34:18.91
2018-10-01 14:34:19.183: Run 103 completed in 00:00:15.108:
2018-10-01 14:34:19.183: Queue q1: 3/3 (100%) queries completed; 3/3 (100%) subqueries
completed; 0 errors.
```

Alternatively, you can use the “LIST” option of the queue statement to use a file that defines the start time of each query to replay queries exactly as they executed in production. This type of testing is used to validate that a platform has the capacity to match the production requirements. It is not a technique to measure the maximum capacity of a platform or comparing several DBMS platforms. Example:

```
queue finance list scripts/finance.lst
```

Where the finance.lst file has the lines:

```
scripts/finance/query001 0
scripts/finance/query002 1.5
scripts/finance/query003 2.7
scripts/finance/query004 4.9
scripts/finance/query005 6.3
…
```

Make sure you have enough workers to be able to execute the queries at the time specified. If all workers were busy when a query was scheduled to execute, since the scheduled time had passed, the query will execute immediately. At the end of the test, a time deficit will be reported which is the time that queries waited for available workers to execute the queries.

Paced Arrival

For proof of concepts, there may be a design goal based on how frequently queries in a queue might be executed. For example, in an airline reservation retrieval application, you might believe that the peak load will be 1,000 reservation queries per hour and 3000 ticket queries per hour. You can approximate an arrival rate using the PACE command to regulate how often a query from a queue will be processed.

Example:

```
Define airpoc simulate reservation system workload
queue res scripts/reservation/*.sql
param res scripts/reservation/passenger.param
pace res 3s
worker res mydb(res_user01:10) 10
queue tkt scripts/ticket/*.sql
param tkt scripts/tickets/ticket.param
pace tkt 1s
worker tkt mydb(tkt_user01:10) 10
run
```

Several points to note about the above example:

1. The POC specification implied a 3:1 arrival rate of ticket queries to reservation queries. More specifically they said the arrival would have been 3000 per hour for tickets or every 1.2 seconds and 1000 per hour for reservations or every 3.6 seconds. The PACE commands used would cause queries to arrive more frequently.
2. For a realistic simulation of a reservation system, we ran queries against the database to create parameter files of reservation keys and ticket keys so on every execution, we would be simulating different passengers stored in different parts of the database
3. We introduced the options on the worker statement to override the user ID specified in the DB alias command with a range of logon ID’s.

Paced Percentage

The goal of simulating a production workload would be to have a mix of short/medium/long queries that mimic the actual production requirements. One customer attempted to choose between two DBMS platforms by running a test of the same tactical query 500 times and then 1,000 times in 500 and 1,000 concurrent sessions. The chosen database chosen based on that test failed to meet the real production requirement because the test was unrealistic. It is important to analyze the mix of queries to determine appropriate ratios of each query type that the platform must support.

In another case, a vendor's workload management prioritized 30,000 general ledger queries to execute in a 1-hour test since it gave a fast path for tactical queries. The customer recognized that far exceeded the demand for that type of query even in a whole year.

To address this, a percentage can be applied to queues based on your analysis of the actual query mix requirement.

```
define EOM_workload Simulate the end of month workload
queue long scripts/long/*.sql
queue medium scripts/medium/*.sql
queue short scripts/short/*.sql
queue tactical scripts/tac/*.sql
worker long mydb(acme_long) 2
worker medum mydb(acme_med) 4
worker short mydb(acme_short) 4
worker tactical mydb(acme_tac) 2
pace short 30%
pace tactical 15%
run 15m kill
```

This defines a total of 12 workers across 4 queues. There are 4 different logon IDS used to override the username in the DB statement, but all use the same URL and password.

There are 2 workers assigned to the tactical queue. When one of the workers servicing the tactical queue requests the next query to execute, if the percent of queries executed in the tactical queue is more than 20% of the total queries executed, the tactical worker(s) will wait until other queues catch up.

The 4 workers assigned to the short queue will also pause starting new queries until the short query count is less than 30%.

If you apply percentages to all queues, TdBench will adjust them to total 100%. If you specify percentages to all queues and one queue struggles to meet its percentage, TdBench will be unable to saturate the platform since the other queues will be waiting. When measuring the capacity of a platform, you should leave one queue without a pace percentage so TdBench can use that queue to use up the rest of the platform capacity.

Running Other Tools During a Test

If you precede the line being queued with the string “os”, it will be executed as an OS command by the worker. Normally queues would contain either all SQL or all OS commands. While a queue with all OS statements can have a worker that uses “OS” as the “database alias”, if the queue contains SQL statements as well, it must have a database alias you defined that connects the worker to a DBMS.

The queue of OS statements can be useful for running data loading activities. Any tool that supports execution from a command line could be used. In the preparation of benchmarks, we often need to load hundreds of tables. To load them quickly, we often use the query driver with multiple workers to load multiple tables in parallel. This can also be used to collect statistics on multiple tables in parallel.

Some benchmarks require the execution of reports via MicroStrategy. MicroStrategy documentation says that the Command Manager 9.x and newer has a command line interface. The example they give is:

```
cmdmgr -n "Project Source Name" -u Username -f "Input File”
```

Based on that documentation, the running of reports from TdBench 8.0 plus an ETL workload would be:

```
Define mstr_etl Workload with Microstrategy and ETL workload
queue mstr cmdmgr -n “Project Source Name” -u UserName -p ${mypw} -o temp/mstr.log
worker mstr os
queue etl scripts/etl/*.sh
worker etl os 2
run
```

When the OS command is executed, the current directory will be the home directory where TdBench was installed. The TdBench CD command does not change the home directory for the OS script. If you need to change directory, include that in your script.

While TdBench captures information about each query that it executes, when you run OS commands, there won’t be any more detail than “it ran” with a start and stop timestamp and return code. The useful output will be in logs that may be written to files plus anything that database query logging can capture.

It is strongly recommended to direct the output from OS commands to a directory and then use the TdBench ZIP command to collect those logs by RunID. The ZIP command should be included in your tdbench.tdb file. Example:

```
zip temp logs erase
```

Result of the ZIP statement:

```
C:\TDBench\temp will be zipped to C:\TDBench\logs directory when RUN command completes.
Directory C:\TDBench\temp will be erased when zip file is successfully created.
```

After the run is complete, you’ll see a message like the following:

```
Zip file C:\TDBench\logs\0107.zip created with all contents of C:\TDBench\temp
Directory C:\TDBench\temp erased after run 107.
```

Environment Preparation Commands

Your testing may require some setup in the DBMS or in the OS environment prior to each test. Examples:

- Frequently we will have tables of different sizes or with different tuning options to be tested. Some tables will be common across all tests (e.g. dimension tables) and others will be a different extrapolation size or have partitioning or indexing differences. A set of SQL commands can be stored in a file to be executed to switch views to point to a different version of tables.
- ETL affects the contents of tables. For consistent results, they need to be reset between tests. The reset scripts could either delete contents from the tables and re-insert from a backup, delete inserted rows, or drop partitions containing new data.
- A queue table used to drive a process needs to be emptied and re-populated prior to the start of a test.
- An ETL process moves input files from “new” to “processed directory. Prior to the next test, OS commands need to be executed to move them back.

The SQL and OS commands are immediate commands and not part of the test. While they can appear anywhere in the script, it is best not to put them between the DEFINE and RUN statement because the execute immediately and the queued SQL and OS commands execute within the test. Someone reading the script might be confused if they appear within the test definition.

The OS command displays output on the console by default or optionally saves the output to a file. If the file name specified to save the data is not fully qualified, it is prefixed by the current TdBench CD setting. However, like the execution of OS commands by the workers, the OS command is executed with the default directory being the home directory for TdBench. Example:

```
os file:temp/os_output.txt dir
```

Result of the OS statement:

```
OS command ran successfully.
```

In Windows, if you make a file reference on the command line with a path, you must follow the Windows convention of using a double back slash \\ to separate the path name parts.

The SQL command has greater functionality. While its default is also to return information to the console, columns will be delimited by the vertical bar | and preceded by a line of column names which makes it useful for copy/pasting to Excel. If outputting to a file for use by a load utility, you will want to skip the first row unless your load utility recognizes embedded column names in input data.

The following is an example of output to the console:

```
sql sb00 select runid, starttime, testname from acme_benchmark.testtracking order by 1;
```

Result of the SQL statement:

```
Line#|RunId|StartTime|TestName
1|1|2017-09-20 09:56:18.07|checkout
2|2|2017-09-25 09:27:06.33|serial
3|3|2017-09-25 16:45:36.86|serial
4|4|2017-09-25 17:13:52.08|WkLd05
5|5|2017-09-25 17:43:23.02|WkLd10
```

```
6|6|2017-09-25 18:08:44.51|WkLd20
SQL command executed 1 subqueries successfully with 0 errors and 6 rows written to
console.
```

There are 2 other output modes:

- TEXT: will output just the first column of text without delimiters or column headings. This is useful for creating scripts dynamically.
- APPEND: same as TEXT but will append and not replace the specified file.

Example of SQL to output text:

```
sql text:temp/table_drop.sql mydb select 'drop table test_benchmark.' || trim(tablename) || ';'
from dbc.tables where databasename = 'test_benchmark' and tablekind = 't' order by 1;
```

Console output from output of TEXT (note that / above was automatically changed to \ ):

```
SQL command executed 1 subqueries successfully with 0 errors and 3 rows written to
C:\TDBench\temp\table_drop.sql
```

File contents written to drop_table.sql:

```
drop table test_benchmark.OneRow;
drop table test_benchmark.TdBenchInfo;
drop table test_benchmark.TestTracking;
```

To verify the output, the command will depend on the operating system where TdBench is running:

```
os type temp\\drop_table.sql          (Windows) … or
:view temp\\droptable.sql
```

```
os cat temp/drop_table.sql            (Linux) … or
:view temp/drop_table.sql
```

You could then execute the created statements in the file by issuing :

```
sql mydb temp/drop_table.sql
```

For more complicated SQL, you can use the delimiter specification to combine multiple lines into a single SQL statement. The Example TEXT sql command could have been:

```
sql text:table_drop.sql mydb delim=eof
select 'drop table test_benchmark.' | | trim(tablename) | | ';'
from dbc.tables where databasename = 'test_benchmark' and tablekind = 't'
order by 1;
eof
```

If the script you are preparing has multiple lines to start and multiple to finish with the core contents extracted from the DBMS, you can use the echo statement with the TEXT/APPEND specification and delim= to write those lines as well. Example:

```
Echo text:newtable.sql CREATE SET TABLE test_benchmark.TestTracking , delim=endtop
    NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
    (
endtop
```

Then follow the above with SQL statements to create the columns using the APPEND and an ECHO statement with the append option to add the final statements to the script. .

Preparing the Queries

A key design criterion of TdBench has been the ability to analyze each query's performance across different tests to determine the impact of changes to the physical model, the scale of data, the number nodes, or the level of concurrency. Without that specific query identification, it is time consuming to determine why the 1000 queries ran longer on this test versus the previous test.

Script Files

TdBench 8.0 will capture the name of the file containing a script as the default name. If a file contains multiple queries, they will be given a subquery number starting with 1, 2, etc and will be in the internal H2 TestResults table.

The identification of each query and subquery are available as variables that can be used in the BEFORE_QUERY and AFTER_QUERY statements, for example, to set a Query_Band in Teradata. Caution: If your test has lots of very short running queries, other queries that are executed before or after the query you are testing will reduce the queries per hour. The use of something like the Query_Band statement should be limited to benchmarks with longer reporting or data mining queries.

Query Naming Via Comments

You can modify the queries to include a specifically formatted comment string. The advantage of this is that the comment will automatically pass thru to the DBMSs query logging, allowing coordinated analysis of query execution from TdBench's internal H2 database and the query logging on the target DBMS (Teradata or others). If the specifically formatted comment string is present in a script file, it will override the naming of the query based on the file name. In the case of Teradata, there are reporting views that automatically parse the comment string to get a query name and query step number. Both analysis from TdBench's internal H2 database and from the DBMSs query log would be in sync.

For SQL, the string is /* tdb=queryname.subname */. Example:

```
Select /* tdb=qry021.02 */ inv_dt, inv_num, ship_from, …
```

In this case the QueryName is "qry021" and the QueryStepName = "02

For an OS command, the tdb= string needs to be in a comment. Examples

```
cmd /c scripts\etl\etl01.bat &:: tdb=etl01          (Windows)
scripts/etl/etl01.sh # tdb=etl01                    (Linux)
```

Queries in Stored Procedures & Macros

If the query is a stored procedure or macro, the distributed reporting package for Teradata will recognize the name of the query from the name of that procedure or macro if the end of the databasename containing that query is "…benchmark. Example:

```
Call acme_benchmark.monthend( … );
exec acme_benchmark.monthend(   );
```

Queries in The Script

You can include the SQL query or OS command on the QUEUE statement. If the line on the queue is a file name or file name pattern, file(s) are put onto the queue otherwise the string on the queue command is put onto the queue. Example:

```
queue q1 select current_date; select current_time;
```

You can also have a multi-line query with the string delim=xxxx at the end of the queue statement where xxxx is a string on a line by itself following the query. For example:

```
2018-10-02 07:30:27.881: define test
2018-10-02 07:30:27.883: Test test defined as runid 111.
------------------- Test: test  Description:  -------------------
queue q1 select time; delim=eof
select date;
select current_time;
eof
2018-10-02 07:31:26.621: queue q1 select time;
select date;
2018-10-02 07:31:26.623: Created q1 with 1 queue entries
```

For the above example, the queryname would be null. You could add a /* tdb=*queryname* */ to the queries to provide a name.

Queue Options

The QUEUE statement can specify when that query or SQL script should run. Example:

```
queue  q1  at 5m  scripts/awful_query.sql
```

The QUEUE statement can queue 1 query. Example:

```
queue  q1  select current_time
```

The QUEUE statement can queue 1 SQL script. Example:

```
queue  q1  scripts/report1.sql
```

The QUEUE statement can queue a pattern of queries. Example:

```
queue  q1  scripts/finance*.sql
```

The queue statement can specify a file of pipe delimited parameters for a single query with each delimited token substituted for :1, :2, :3 …. Example:

```
queue  q1  scripts/call_center_rpt01.sql  param  scripts/call_center_rpt01.param  ‘|’  *
```

The QUEUE statement can specify a list of queries with start times

```
queue  q1  scripts/sales_query.lst
```

Where scripts/sales_query.lst contains

```
Scripts/sales_rpt01  3.5s
Scripts/sales_rpt02  4.3s
```

Use HELP QUERY to see even more forms of the QUEUE statement.

Artifacts and Analysis

One key responsibility during a benchmark is to collect data for analysis and proof of the results of the testing. Artifact collection and rapid, productive analysis has driven the development of TdBench over the past 14 years. Without a central tool that is being used by all analysts in a benchmark:

- the logs from various tools will be helter-skelter across servers, PC, and directories as various analysts follow their own design and naming conventions
- there will be no systematic identification of test runs except thru diligence and effort by some test coordinator
- As tuning changes are made to workloads with thousands of queries, it will be difficult to decompose the results to identify which got better and which got worse
- The analysis of results may be deferred until after the equipment is no longer available because of the rush to complete testing and flaws will be detected on tests that should have been fixed and re-run.

TdBench 8.0 extends this capability to multiple DBMSs. The H2 database built into TdBench:

- identifies each test with a RunID, name, start/stop times and allows notes,
- captures the statements and messages during the setup of the test, and
- captures the results of each query execution as viewed by the client.

There is also an interface that can be used to coordinate a test tracking table in the DBMS that captures start/stop time stamps in the time zone and precise clock that is used to record query logging and system utilization information. This information can dive deeper into the "why's" of query and workload performance.

Finally, there is a facility that will collect the contents of a directory containing logs from various external tools executed during a benchmark and zip them up into a file coordinated with the RunID of each test.

The H2 Database of Results

The H2 database is a light weight, open source SQL database for Java that is used to store the results of tests. You can use fairly standard SQL to perform analysis. A manual is available at https://github.com/h2database/h2database/releases/download. (current release can be found by issuing "select H2VERSION()". You can compose your favorite analysis in command files that you run with the INCLUDE command. For example, you could create stdrpt.tdb:

```
select queryname, count(*), avg(RESPTIMEMS), sum(case when returncode = 0 then 0 else 1
end) from testresults where runid = :i1;
```

and run it by:

```
include stdrpt.tdb 21
```

or use the alias to INCLUDE:

```
exec stdprt.tdb 21
```

Results come out delimited to the screen or you can export the delimited results to a file. In one benchmark during the Beta testing of TdBench 8.0, the customer requested the vendors submit the H2 database with the results of the test so the customer could do their own, unbiased analysis.

TestTracking table in H2

Information for this table is maintained in memory during the run and written to the H2 database at the end of the run. The RunID is always 1 greater than the max(RunID) of the tests in the table. There may also be a TestTracking table on the target DBMS with a different RunID. This table's columns:

RUNID	Automatically assigned starting with 1
TESTNAME	First word from the DEFINE statement
TESTDESCRIPTION	The rest of the line on the DEFINE statement
STARTTIME	Start time of the RUN, according to the client Server/PC clock
STOPTIME	Stop time of the RUN, according to the client Server/PC clock
LOGCOUNT	Count of lines written to TestTracking Log
RESULTCOUNT	Count of lines written to TestResults; implies # subqueries
ERRORCOUNT	Count of subqueries with a non-zero return code
NOTES	Text you insert or append with the TdBench NOTES command.
URL	Identifies which server this test was run against.

You can easily see output from this table with the LIST command that has the following options:

list	Lists out all tests
list -5	Lists the most recent 5 tests
list 21	Lists out just information on RunID 21
list 31 35	Lists out RunID's 31, 32, 33, 34, 35
list where testname = 'test01'	Lists out RunIDs with the given test name
List where notes like 'final%'	Lists out RunIDs you've flagged as final (see below)

The columns listed by the LIST statement are:

RunId
TestName
TestDescription
StartTime
EndTime
#Log Entries
#Results
#Errors
Notes

Next, a list of any archive prefixes is provided.

NOTE Comand

You can add or remove notes with the NOTE command. You may find that you'll run hundreds of tests during a benchmark. Some will have different conditions (e.g. tuning, data volumes), or you may observe there were mistakes in setup. At points during the benchmark you may want to identify a run as a "final" run to include in your presentation. The following are examples of the NOTE command:

note 75 recollected stats on fact table	Sets the text into TestTracking notes for Runid 75
note 75 add -usethis	Puts -usethis at end of note.
note 75	Removes the note from RunID 75

You can use SQL to query the table. By default, SELECT goes against the H2 database. Examples:

```
Select runid, testname, starttime, stoptime, notes where notes like ‘%-usethis%’;
```

```
Select runid, testname, TimeStampDiff(‘SECOND’, starttime, stoptime), resultcount, errorcount from testtracking where testname like ‘wkld%’ order by testname;
```

The AFTER_NOTE command can be put in tdbench.cmd to update the note in the host DBMS. Example:

```
after_note sql ${TdBenchServer} update ${TdBenchDb}.testtracking set runnotes=':note' where runid = :runid;
```

TestTrackingLog

This table in H2 lists out all of the input statements, informational messages, and error messages that were issued during your test. It has the following columns:

RUNID	Automatically assigned starting with 1, *same as TestTracking table*
LOGLINENO	Sequence number of the lines
LOGTIMESTAMP	Time the command or message occured
LOGTYPE	Either: Input, Error, or Info (Case Sensitive)
LOGTEXT	Text of the command or the message

The lines are posted to the table at the completion of every RUN command. As a result, any preparation statements that you execute like SQL or OS, and for the first test when TdBench is initiated, any lines from the tdbench.tdb file will be shown in the TestTrackingLog table for that RunID.

You can use standard SQL against this table. Example:

```
select tt.runid, tt.testname, ttl.logtext delim=eof
from testtracking as tt join testtrackinglog ttl
where logtext like '%shipqry%' and logtype=’Input’;
eof
```

You can also use one variation of the SAVE command to "Save" a RunID's commands so you can edit and execute them again. Example:

```
save 31 scripts/wkld05.tdb
```

TestResults

This table in H2 has one line from every query or OS statement executed during a test (not from the SQL or OS setup commands). It has the following columns:

RUNID	Automatically assigned starting with 1, *same as TestTracking table*
QUERYID	Query or command number within the queue (starts with 1)
SUBQUERYID	Subquery or command number within a query (starts with 1)
QUEUENAME	Name of the queue
WORKERID	Worker number for the queue (starts with 1)
USERNAME	User name for the worker
QUERYNAME	Name of file or comment containing tdb=xxxxxxxx
RETURNCODE	return code from the subquery or command
ERRORMSG	Text of the error that occured
STARTTIMESTAMP	Time the query or OS command started
RESPTIMESTAMP	Time the query or OS command responded
RESPTIMEMS	Number of milliseconds between the two fields above
ROWS	# of rows returned by the subquery if ROWCOUNT was executed
DEFICITMS	Milliseconds waiting for workers to come free when using PACE
PARAMS	Parameter line associated with this query's execution

In addition to the SQL statements against TestResults, there is a DELETE statement that can remove the hundreds of thousands of rows from TestResults when a test goes horribly wrong (like failing to set up access rights). It only deletes rows from TestResults and appends a comment to TestTracking. Example:

delete 74	Will remove all rows from TestResults table in H2 with RunID 74.

Reporting Analysis Views

Views are available to simplify the analysis of runs. You should always include a WHERE clause and an "ORDER BY columname1, columnname2 …" clause to sequence the output. (H2 doesn't support "order by 1,2,3...”). The data for these views come from a join between TESTTRACKING and TESTRESULTS.

RPT_TESTS

This view provides a summary by test at a slightly lower level than available from the LIST command. The columns in that view are:

RUNID	*group by*
TESTNAME	*group by*
STARTTIME	*group by*
STOPTIME	*group by*
RUNSECS	*group by*
NUMEXEC	*calculated*
NUMNOERROR	*calculated*
NUMQUEUES	*calculated*
NUMWORKER	*calculated*
AVERESP	*calculated*

RPT_QUEUES

Often queries of different types are organized into queues, such as sales queries, distribution queries, accounting queries, update queue, etc. The columns in that view are:

RUNID	*group by*
TESTNAME	*group by*
QUEUENAME	*group by*
NUMWORKER	*calculated*
NUMEXEC	*calculated*
NUMNOERROR	*calculated*
AVERESP	*calculated*
MINRESP	*calculated*
MAXRESP	*calculated*

Example: *Retrieve summary of a given queue across tests you labeled with a note starting with “final…”*

```
Select * from rpt_queues where notes like ‘final%’ order by testname, queuename;
```

RPT_QUERIES

This view summaries the execution of each query by name.

RUNID	*group by*
TESTNAME	*group by*
QUEUENAME	*group by*
QUERYNAME	*group by*
NUMQUERY	*calculated*
NUMEXEC	*calculated*
NUMNOERROR	*calculated*
AVERESP	*calculated*
MINRESP	*calculated*
MAXRESP	*calculated*

RPT_ERRORS

This view summarizes errors that occur in a test. It is a good idea to query this early in a project until you get all of the access rights and query syntax cleaned up.

RUNID	*group by*
TESTNAME	*group by*
QUEUENAME	*group by*
QUERYNAME	*group by*
RETURNCODE	*group by*
ERRORMSG	*group by*
NUMERRORS	*calculated*

Archiving, Dumping, and Restoring Sets of Runs

When TdBench is used in competitive benchmarks, it is desirable to analyze the results across vendors. A set of commands is provided to create a set of test results, export them to a zip file, then import them to the same or another database.

Each TdBench installation has the following objects that have been previously described. When an archive is created, the standard objects get a prefix. Example:

Standard Object Name	Examples of archive object names	
TESTTRACKING	IBM_TESTTRACKING	SNOWFLAKE_TESTTRACKING
TESTTRACKINGLOG	IBM_TESTTRACKINGLOG	SNOWFLAKE _TESTTRACKINGLOG
TESTRESULTS	IBM_TESTRESULTS	SNOWFLAKE _TESTRESULTS
RPT_TESTS	IBM_RPT_TESTS	SNOWFLAKE _RPT_TESTS
RPT_QUEUES	IBM_RPT_QUEUES	SNOWFLAKE _RPT_QUEUES
RPT_QUERIES	IBM_RPT_QUERIES	SNOWFLAKE _RPT_QUERIES
RPT_ERRORS	IBM_RPT_ERRORS	SNOWFLAKE _RPT_ERRORS

ARCHIVE

The ARCHIVE command can have a WHERE clause to allow you to assemble subsets of the test results from the standard object tables to the archive. The WHERE constraints are applied against the columns in the TESTTRACKING table and corresponding rows from TESTRESULTS and TESTTRACKINGLOG are put into the archive tables prefixed by the given name along with the TESTTRACKING rows. This is done with a MERGE so you can add rows to an archive or use the NOTE command on the standard TESTTRACKING table and archive again to update the ARCHIVE rows. A set of reporting views is also created for each object. Examples:

Command	Results
archive final where notes like 'final%'	All tests that you've labeled as final are in a set of tables prefixed by "final" and a set of reporting views created are prefixed by "final"
archive final where runid in (21,22,24,26)	The tests listed are merged into the "final" archive, inserting if not previously there or updating (generally only notes) if previously saved
Archive serial from final where testname like 'serial%'	Instead of pulling rows from the standard table, rows are pulled from the "final" archive.

DUMP

The DUMP command takes a completed archive, exports it as 3 CSV files and puts them into a zip file prefixed by the archive name or another name you supply. Examples:

Command	Results
dump final	The contents of FINAL_TESTTRACKING, FINAL_RESULTS and FINAL_TESTTRACKINGLOG are put into final_archive.zip.
dump final to results_2019_02_07	Same as above, but the resulting zip file is name is: results_2019_02_07.zip

RESTORE

The restore command will restore the 3 tables with results from a zip file and optionally rename it. It will create the target tables and associated reporting views before merging in the data from the zip file. If rows with the same RUNID exist in the target archive, they will be updated, otherwise they will be inserted.

CAUTION: If you restore tests from different instances of TdBench, they may have overlapping RunIDs and the result will be a corruption of results in the target archive.

Examples:

Command	Results
restore final	The zip file is restored to FINAL_TESTTRACKING, FINAL_TESTRESULTS and FINAL_TESTTRACKINGLOG DO NOT USE a generic name like "final" if getting "final" results from multiple vendors
restore final to teradata	Creates and restores data to TERADATA_TESTTRACKING, TERADATA_TESTRESULTS, and TERADATA_TESTTRACKINGLOG.

Reporting from Archives

The LIST command allows the archive name to be passed as the first token. The full syntax of LIST is:

```
list {archivename} {-count | runid {to-runid} | "where" where-condition}
```

Examples:

Command	Results
list ibm	Lists the tests from the tests submitted by IBM showing RunID, Testname, number of queries, number of errors and notes.
list snowflake where testname like 'wkld%'	Lists out the tests having a name beginning with wkld from the snowflake archives
list oracle -5	Lists out the last 5 tests from the oracle archive
list greenplum 20 25	Lists out the RunID's from the Greenplum archive between 20 and 25
list 20 25	Lists out the RunID's from the standard tables plus the names of available archives
list 0	… tricky – Since RunIDs begin with 1, no rows from the standard tables will be listed, however the archive names will be shown.

To analyze from the reporting views, use the UNION operator in your select statement. Example:

```
Select 'ibm' as vendor, rq.* from ibm_rpt_queues as rq where rq.testname like 'wkld%' union all
select 'oracle' as vendor, rq.* from oracle_rpt_queues as rq where rq.testname like 'wkld% union all
select 'teradata' as vendor, rq.* from teradata_rpt_queues as rq where rq.testname like 'wkld%
order by vendor, testname, runid, queuename;
```

Note: You would either need to put the above select statement into a file and execute it or suffix each line with a + indicating the next line is part of the current command.

Connection to DBMS Query Logs

Earlier versions of TdBench worked only with Teradata, in fact, until release 6, the queues of work were Teradata Queue Tables. The idea of a Test Tracking table has been present from the start. Reporting views and macros against the Teradata DBQL and Resusage tables were driven by joins to the TestTracking table which contained the Start and Stop timestamp of each RunID.

In Release 7, we moved the maintenance of the TestTracking table into external macros that allow the TdBench reporting to be used with other tools by wrapping their execution between the execution of TestStart and TestStop. Both macros not only recorded the timestamps, but automatically recorded the beginning and ending system workload, nodes online before and after the test, and the workload settings.

For TdBench 8.0, to continue to leverage that reporting to investigate details behind each test's performance, we created the BEFORE_RUN and AFTER_RUN commands that are SQL or OS commands to be executed before the RUN statement and After all of the processing in the client of the test is complete.

For any DBMS that has query logging with details of the CPU, I/O, and steps at the query level, or system level CPU, I/O, and system workload, here are some suggestions on how to use it.

BEFORE_RUN

This statement could execute an insert on the DBMS using the TdBench Variables :runid, :runsecs, :testname, :testdescription, and:workers to create a row in a TestTracking table on the DBMS that includes the DBMS TimeStamp when the row is created. Or,

AFTER_RUN

These statement could Update the row in the TestTracking table matching the :runid variable with the Current_Timestamp, :resultcount, and :errorcount.

IMPORTANT: While the BEFORE_RUN and AFTER_RUN statements can be processed at any time during a TdBench session, the AFTER_RUN won't work if it is after the RUN statement. They are intended to be executed once per session and apply to all runs. it is best to put them in the tdbench.tdb file once you have tested them because each time you execute them, they add to the existing BEFORE_RUN and AFTER_RUN statements so if they are in a test script that you execute multiple times, you'll start repeating your statements more times each time you test. There is a "delete" option on the commands to clear the stored BEFORE_RUN and AFTER RUN statements.

Reporting Views:

For the DBMSs Teradata (a.k.a. **V**antage **C**loud **E**nterprise VCE) and Teradata_Lake (a.k.a. **V**antage **C**loud **L**ake or VCL), there are extensive reporting view installed via the SETUP command.

- VCE has reporting views against DBQL and ResUsage in DBC unioned with PDCR or prior DBC releases.
- For VCL, there are reporting views against DBQL and ResUsage in TD_METRIC_SVC.
- For a few other databases, there are views that join TestTracking to the query log to allow simple constraints by RunID. If the DBMS log retention was short, there is code to create a history of log rows that occurred between the start and stop timestamps in TestTracking.

Automatically Zipping Test Artifacts

If you run other BI, ETL, or other packages as a part of your testing, to get those saved for reference along with a test, the TdBench ZIP command can be used (usually in the tdbench.tdb) to set your preferences for the standard directory where your temporary output will be placed and where you'd like the zip file to be placed.

On install, the following directory structure will be setup below the home directory of Tdbench:

qdriver	Or whatever name you gave to the TdBench home directory
logs	Target directory to hold the zipped logs for each RunID
temp	Put all files you want to preserve here
setup	Directory for setup scripts
teradata	Directory containing setup scripts for Teradata
scripts	Suggested place to put your TdBench scripts

In your tdbench.tdb file located in your TdBench home directory (above: qdriver), you would put:

```
zip temp logs erase
```

Then after every run, if there are files in the temp directory, they will be zipped up into the logs directory. For example, for RunID 75, the artifacts of your test from the temp directory will be put into a file name temp/0075.zip.

For example, if you had a shell script that you used for table loading and you had a parameter file with the table name and file name delimited by the vertical bar, if your script and utility output results to standard out, the statements for that queue would be:

```
queue loader os scripts/loader.sh :1 :2 > temp/:1.log  2>&1
param loader scripts/loader_filelist.txt |
```

The first line above puts a shell script on the loader queue. It will be passed 2 parameters and its output will be directed to a log file in the temp directory with a parameterized name and a file suffix of ".log".

The second line with define a list of parameters for the loader queue. The contents of that parameter file could be:

```
Region|region.txt
Customer|customer.txt
Lineitem|lineitem.txt
…
```

You could then specify a worker statement for the loader script that specified multiple OS workers to perform data loading in parallel:

```
worker loader os 2
```

All output from the loading would be written to temp/REGION.log, temp/Customer.log, temp/LineItem.log … and when all loading is complete, the temp directory would be zipped up to logs/nnn.zip where nnn is the test's RunId (corresponding to the TestTracking table).

Advanced Scripting

REPLICATE, SHUFFLE, SAVE

The easiest way to invoke queries is to have each query in a directory and use a wild card search to include those queries into a directory. In some cases, the queries may be in multiple directories and perhaps you want to make a larger number of queries, perhaps set a ratio of different queries. Below are a couple of examples and explanations demonstrating the use of these commands:

queue q1 scripts/queries/mstr*.sql queue q1 os scripts/queries/elt*.sql replicate q1 3	Would create 3 sequences of Microstrategy queries followed by ELT queries.
queue q1 scripts/finance/*.sql queue q1 scripts/sales/*.sql queue q1 scripts/sales/*.sql queue q1 scripts/marketing/*.sql shuffle q1 save q1 scripts/combined/qry*.sql	This would put1 copy of light, 2 of medium and 1 of the heavy command, then randomize them. This is then saved to a new queue in the combined directory with names qry001.sql, qry002.sql …

It is a good idea to save queues after you shuffle them so that on each execution, you get the same random pattern for repeatability.

Clone

This command is similar to REPLICATE but it will copy all queries and workers associated with a queue. This enables simulating TPC and Jmeter typical benchmarks where multiple workers run the same queries in the same sequence. While this doesn't create a realistic benchmark, it allows comparison of benchmark approaches to hopefully argue for a more realistic workload design. Example:

```
replicate q1 q2
```

Parameter Files (PARAM Command)

Production databases don't just answer one question for one user. A good benchmark has an assortment of queries that represents the same type of workload that occurs in production. However, even with due diligence to sample queries out of production, you may pick up query for just one customer that would normally be applied to multiple customers in a normal day. The easiest way to do this is to replace the literals with variable markers and use the PARAM command to change the variables on each execution of a query.

There was an example of this in the discussion of zip files. Below is another simple example. First, create a parameter file from the database:

```
sql text:scripts/500customers.txt mydb select top 500 o_custkey from tpcd.orders group by 1;
```

Then use that in setting up a queue

```
queue q1 select o_cust_key, sum(o_total_price) from tpch.orders where o_custkey = :1;
param q1 scripts/500customers.txt
```

The above applies the PARAM to the queue. You can also put PARM on individual queries.

The INCLUDE Statement (Alias: EXEC)

The INCLUDE statement allows you to reuse groups of TdBench commands either by themselves or used in other command files. For example, you might have a complicated setup for a group of queues and would like to use that setup in different commands. An even more powerful usage is with parameters to reduce the number of scripts you need to create for a benchmark.

You could build some analysis queries to run against H2 and then include them passing the RunID as a parameter for both a constraint and as part of the output file name. (See example in The H2 Database)

The parameter markers for the include statements are of the form :i1 :i2 :i3   (to distinguish them from PARAM which use :1 :2 …).

For example, you might create the file scripts/wkld.tdb with the following lines:

```
Define wkld:i1 Workload test for :i1 users for :i2 minutes
queue q1 scripts/custdollar/*.sql
worker q1 mydb :i1
run :i2m kill
```

You could then run a series of test Either from the command line or from a command file:

```
include scripts/wkld.tdb 05 30
include scripts/wkld.tdb 10 30
include scripts/wkld.tdb 20 30
include scripts/wkld.tdb 40 30
include scripts/wkld.tdb 80 30
```

Variables

You've seen examples above using variables in numerous ways. For the current variables in the release of TdBench you are using, use HELP VARIABLES.

The types of variables are listed below:

Type	Examples	Source	When Processed
INCLUDE	:i1 :i2 :i3 …	On the INCLUDE command	As each line is read from the include file. They can be in comments or quoted literals.
Environment	${HOSTNAME} ${LOGNAME) ${PWD} ${HOMEPATH} ${USERNAME} ${MyPass} ${scale}	Linux Linux Linux Windows Windows You set You set	After each line is read from batch command files on the TdBench command line, from include files, or from the console and before they are parsed, these substitutions are applied. They may be inside comments or quoted literals. You can even have entire commands placed in a variable.

Type	Examples	Source	When Processed
Program Variables	:runid :testname :runsecs :resultcount :linecnt :retcode	During processing of the TdBench Commands	For OS, SQL, ECHO and IF statements. They may appear inside comments and quoted strings.
PARAM variables	:1 :2 :3 …		By the worker during the test. Workers take the next PARAM row coordinated across workers. They may not appear inside literals due to confusion with time literals. If using a character literal for a PARAM, enclose in quotes on the PARAM file
Worker Variables	:runid :testname :queryname :queue :timestamp		By the worker during the test after the PARAM values have been substituted but before the query executes. They may be in comments or literals. For example, the timestamp would be the client clock's time of query initiation.

IF, GOTO and LABEL

TdBench 7.x introduced these commands for automating scripts to detect failures and abort execution. TdBench 8.0 was faced with having to run on multiple client platforms and setup could be more of an issue. These statements support very simplistic logic to skip down the script based on tests of variables. There is no looping support.

The IF statement is:

IF *x* *condition* *y* THEN *statement*.

Either side of the condition (x or y) may be:

- A literal without spaces. Sorry, quotes won't help
- A program variable starting with colon … : … sorry, you can't build complex string
- An include variable – substituted through the command when first processed
- An environmental variable – substituted throughout the command when first processed

The condition may be: =, !=, >, >=, <, or <=. If both x and y are numbers, a numeric comparison will be done, otherwise it will be an alpha comparison. There is also "is set", "not set", "exists", "not exists' and "begins" comparison operator. that tests if x begins with the string in y. This is useful in INCLUDE scripts to test for omitted parameters. Example:

```
If  t.tdb  not exists then echo the file t.tdb was not found
if :errorcount  >  0  then  goto stop_test
if ${pwd} not set  then echo Can not continue since  environment variable pwd is missing
```

The word THEN must appear to indicate the rest of the line is a statement. This statement could be a GOTO, OS, SQL or even another IF statement which thereby would create a crude "AND" capability. (You would get OR by putting IF statements following each other).

The following is an example from one of the Teradata setup scripts:

```
echo Creating or revising Testtracking tables
sql  ${TdBenchServer} delim=eof
locking row for access select tablename from dbc.tables
where tablename = 'TdBenchInfo' and databasename = '${TdBenchDb}';
eof
if :retcode > 0 then goto dbc_rights_missing
if :linecnt > 0 then goto HaveInfoTbl

sql ${TdBenchServer} delim=eof
Create set table ${TdBenchDb}.TdBenchInfo
  (InfoKey varchar(30),
   InfoData varchar(256))
Unique primary index(InfoKey);
eof
if :retcode > 0 then goto db_missing

label HaveInfoTbl
sql ${TdBenchServer} delim=eof
update ${TdBenchDb}.TdBenchInfo set InfoData = '$TdBenchPrefix'
where InfoKey = 'PREFIX' else insert into ${TdBenchDb}.TdBenchInfo
values('PREFIX','${TdBenchPrefix}');
update ${TdBenchDb}.TdBenchInfo set InfoData = '${TdBenchVersion}'
where InfoKey = 'VERSION' else insert into ${TdBenchDb}.TdBenchInfo
values('VERSION','${TdBenchVersion}');
eof
if :retcode > 0 then goto db_rights_missing
```

The GOTO statement causes TdBench to start skipping lines from the include file until:

1. A LABEL statement is found with a matching label name
2. A DEFINE statement is found with a matching test name
3. End-of-file on the included file

SET, Read

This allows setting string variables that can be used in script logic and formulation of statements. The examples of usage:

```
set user = johndoe
read pass Enter the password for :user:
if :pass not set then echo Unable to execute without a password
if pass not set then goto endtest
read ans=yes Do you want to start the test? Enter yes or no:
if ans != yes then goto endtest

… the test

label endtest
```

This sets a literal for the user's name, then uses that string in the prompt to get a password. If the password isn't entered, control jumps to a label statement at the end of the script. It then prompts to see if they want to proceed with a default of yes if the user just presses the enter key.

You cannot set/change the built-in variables like username, runid, testname, etc. To see the values of all variables currently defined, issue SET by itself. Use "HELP VARIABLES" to get a description of variables and where they are set.

You can use variables like command aliases. The SETUP command for 8.01.03 created :edit and :view variables that are set to appropriate default utilities based on the operating system. Example:

```
set s  =  sql mydb select
```

Then using it:

```
:s  current_timestamp
```

TRACE and ECHO

These statements were mainly put in place for debugging TdBench, but have been found to be useful in constructing the installation scripts.

ECHO is similar to the Linux and Windows statements that its role is to put text out on the system console. It can be a multi-line statement using the delim=xxx specification and it is useful for displaying the current contents of variables. In fact, there is a special variable :all which will display all the defined program variables. Example:

```
Echo :all
```

The normal mode for TdBench is to output every statement as it is executed. With:

```
Trace off
```

Statements will only be displayed if there was an error. Trace by itself will display whether trace is ON or OFF.

You can issue:

```
trace list
```

to list out on the console all lines and messages since TdBench started or since the last RUN statement.

You can also save the list of commands and messages to a file where you can apply search statements with file editors to look for errors or other text. Example:

```
trace save temp/trace.txt
```

You may want to issue “trace clear” at the end of tdbench.tdb to clear the startup statements from the trace buffer so they aren’t saved to testtrackinglog or when issuing SAVE of a runid.

SLEEP *(Not for the Benchmark Analyst)*

Frequently there is a maintenance window where a benchmark can run. The following is an example of waiting until 11 PM and then trying several times to see if the activity sessions will allow the test to start:

```
sleep 23:00:00
sql mydb select * from dbc.sessioninfo;
if :linecnt < 5 then goto serialtest
sleep 5m
sql mydb select * from dbc.sessioninfo;
if :linecnt < 5 then goto serialtest
sleep 5m
sql mydb select * from dbc.sessioninfo;
if :linecnt < 5 then goto serialtest
goto cantrun

define serialtest Execute all queries in single stream
queue serial …
…
run
goto end

label cantrun
echo delim=eof
ERROR – Could not get less than 5 active sessions before initiating test
eof

label end
```

Setup for TdBench 8.01.03 & Later

Background:

Prior to this release, the installation of TdBench, the setup of network connections, databases and users was supported by execution of TdBench with “setup” on the command line which told tdbatch.sh or tdbatch.bat to run versions of Linux shell scripts or Windows batch files in the setup directory. Those scripts provided menus, invoked the platform’s editor and ran the TdBench Java application to connect to the host DBMS to perform the setup. The complexity of the development and support of these scripts was a barrier to others wanting to adapt TdBench to their DBMS.

SETUP is now a command built into the TdBench Java program that reads a platform specific menu file to display steps, then run TdBench scripts to perform the DBMS setup. The list of DBMSs shown from SETUP will either have setup scripts and help files or a readme.txt file providing basic instructions.

Getting the TdBench Distribution Zip File

This file is posted on https://downloads.teradata.com as “TdBench for any DBMS”. Download to a directory on your server or PC. When you unzip the file, you will get the following content:

readme.txt	General usage information (for those not reading this)
release_notes.txt	Recent changes. Full list under Tdbench: HELP version
tdbench.sh	Invocation of TdBench under Linux & others
tdbench.bat	Invocation of TdBench under Windows
Logs	Empty directory to hold setup logs and test artifacts from ZIP
scripts	Empty directory to hold your TdBench Scripts
setup	Installation materials
firsttime.txt	
license.txt	
Directory for each DBMS	Each DBMS has at least a readme.txt and may also have:
setup.menu	Parameters and text for the SETUP command
help	One help file for each SETUP menu option
scripts	Scripts executed as referenced by setup.menu options
templates	Startup OS & TdBench scripts to be copied for your updating
temp	

If you don’t see your DBMS in the setup directory or when you run the SETUP command, updated information/content may be at https://github.com/Teradata/tdbench. There is general instructions below if your DBMS is not listed. **We would appreciate any submissions to the above GitHub site**.

Getting the JDBC driver for your DBMS

DBMSs with setup scripts allow you to issue help to get more setup information including the likely download site for your DBMS’s JDBC driver. Setup will offer to show you the readme.txt for the other documented databases which will provide you manual instructions for setting up your DBMS. Otherwise, use your web browser to search for your DBMS’s download location for their JDBC driver.

Download the driver to the same directory where tdbench.sh and tdbench.bat reside.

TdBench Startup Files

The startup files are designed to easily support multi-DBMS benchmarks but if your needs are simple, you may only need to make changes to one file (or less). The first option in every setup menu will copy up to 4 files from that DBMS's setup directory and then prompt you for connection information.

There are two or more shell script files (either .sh or .bat) that set environment variables and can be used to prepare directories and files when performing repeated ETL tests. There are also 2 or more TdBench script files (.tdb) that define the connection to the DBMS

tdbench_config.?	Sets the TdBench JAR version and calls current DBMS script
tdbench_redshift.config.? *tdbench_xxx_config.?* …	Sets variables used to support setup scripts and can set or prompt for environment variables to be used by OS programs called from TdBench
tdbench.tdb	Can prompt for common variables across DBMS scripts
tdbench_redshift.tdb *tdbench_xxx.tdb* …	Definition of JDBC driver and connection credentials for connecting to your DBMS. Can also setup statements to coordinate with host DBMS reporting

The most important file to review is your DBMS's tdbench_xxx.tdb (where xxx is your DBMS name). You can define connection information there that will be used every time you run TdBench interactively or in batch.

The SETUP Menu

When you type: SETUP, each subdirectory under the setup directory is listed for you to choose. Example:

```
qdriver: setup

 ----- DBMSs defined in the setup directory -----
   1: databricks
   2: google_bigquery
   3: greenplum
   4: ibm_netezza
   5: ibm_sailfish
   6: oracle_exadata
   7: redshift
   8: snowflake
   9: teradata
   10: teradata_lake

Enter a number from the list above:
```

When you select a DBMS, TdBench will restart to load any previously edited startup scripts for that DBMS then offer something like:

```
The DBMS 'databricks' provides help information only.
Enter 'help' or 'quit':
```

Or:

```
The DBMS 'snowflake' provides both help information and setup.menu.
Enter 'help', 'setup', or 'quit':
```

If your DBMS has setup scripts, you'll get a menu something like below:

```
================== TdBench Setup Menu Steps / Options for snowflake ===================
 0: exec copy_templates.tdb - Provides starter set of config files or just setup a DB for logon

 1: :edit tdbench_config.sh - OPTIONAL - customize to choose jar file version and default DBMS
 2: :edit tdbench_snowflake_config.sh - Customize environment variables
 3: :edit tdbench.tdb - OPTIONAL - Initial command file executed by jar file to customize session settings
 4: :edit tdbench_snowflake.tdb - Command file to define class and db connections for Teradata servers

 5: exit 123 reset - IF NEEDED - Restart tdbench to reload the settings from above 4 config & .tdb files

 6: exec test_connect.tdb - Run simple select to validate your logon ID on host DBMS
 7: :edit create_tdbench.sql - Edit the script with create database, table and view statements for host DBMS
 8: exec create_tdbench.tdb - Define the database/tables/views on target Snowflake DBMS

Enter a number from the list above, enter "help" for general information, "help n" for help on an option

Enter a number, "help", "help n"  or "quit":
```

(Italicized options above will vary by DBMS)

The references Comments on the options (suggestions on shortcuts after the list):

0. will execute copy_templates.tdb which will copy the templates to TdBench's root directory if the file doesn't already exist there. (because of running option 0 previously for another DBMS). It will then offer to create a database alias containing URL and credentials for your DBMS that it will save in your DBMS's TdBench script file (e.g. tdbench_xxx.tdb).
1. tdbench_config.? sets the default DBMS in an environment variable ${TdBench_DBMS}. This can be overridden on the TdBench command line with the -d option followed by the database name. If the DBMS specific shell script exists, it will run that as well.
2. tdbench_xxx_config.? sets DBMS specific environment variables that are needed if you are running scripts to setup host DBMS query log reporting. You can also use this to set DBMS specific environment variables that can be used by OS commands run under TdBench (such as passwords)
3. tdbench.tdb can be used for prompting or setting variables for use across DBMSs, and then if the there is a DBMS specific TdBench script exists (as defined by ${TdBench_DBMS}), it will run that script.
4. tdbench_xxx.tdb loads the JDBC driver for your DBMS using the CLASS statement. It then uses the DB statement to define a database alias with the class reference, URL and credentials for you to connect to your DBMS and run your test. This is a good place for other setup statements to define BEFORE_RUN and AFTER_RUN with statements to execute before and after each test.

You can also specify the capture and disposition of log files generated by each test with the ZIP command. **(See individual command descriptions below)**

5. Since the above steps impact the startup scripts supporting your DBMS, you need to restart TdBench so it can read what you changed.

The steps that follow the restart of TdBench will differ by DBMS. There is usually:

6. A test of connectivity by using the database alias to logon to your DBMS
7. Editing of a setup SQL script to define tables in the target DBMS
8. Execution of the setup script

Teradata (a.k.a. Vantage Cloud Enterprise)) and Teradata Lake (a.k.a. Vantage Cloud Enterprise) have additional scripts to set up reporting macros and views and provide useful macros for preparing and monitoring benchmark tests.)

<u>Setup Shortcuts and Hints:</u>

1. For a single DBMS without reporting against host DBMS query logs, just run #4 and #5
2. If there isn't a setup for your DBMS, use option #0 to get a starter set of scripts into the TdBench root. If only using one DBMS, you can just customize tdbench_config.? and tdbench.tdb.
3. You can put -d <DBMS name> on the command line to override the ${TdBench_DBMS} in tdbench_config.?. If using TdBench interactively on one DBMS, you can switch using SETUP to chose the alternate database, then use option #5 to restart using the other DBMS.

Statements for Standard Startup Script

CLASS – Define JDBC Driver

This loads a JDBC driver into TdBench for your DBMS. Syntax:

CLASS [class-name-of-JDBC-driver] [JDBC Protocol] [JDBC driver file name(s) | path search]

All three parameters are required. The CLASS statement for a given class name may only be loaded once. You will need to review the DBMS's JDBC documentation to find the class name and JDBC protocol.

Earlier versions of the Teradata JDBC driver required 2 files so we allow multiple jar file names on the class statement. We found that the JDBC driver for Google BigQuery is distributed as a directory with about 70 files, so we allow specification of a search path which could be a directory. For simplicity, we suggest locating the JDBC driver in the TdBench home directory, but if you already have a copy, you can specify any path name.

Examples:

```
class com.teradata.jdbc.TeraDriver jdbc:teradata terajdbc4.jar

class com.simba.googlebigquery.jdbc.Driver jdbc:bigquery bigquery/*
```

DB – Define Database Alias to Connect to Your DBMS

The DB statement defines the URL and JDBC protocol and optional username and passwords for connecting to your DBMS. You may have multiple DB statements to connect to the same DBMS, either for different workloads or different privileges, such as:

- Different logon IDs for different workloads to be able to easily observe what is running during tests on your DBMS console.
- A special alias with a username having elevated privileges for use interactively while setting up your test.

Syntax:

```
db  [db-alias-name]  [jdbc-protocol]://[IP or URL of server]  {username}  {password}}
```

Examples:

```
Db  tdprod  jdbc:teradata://10.25.11.107/DATABASE=MyDB  MyUser  Xpassword123

db  mydb  jdbc:snowflake://xxxxx-yyyyy.gcp.snowflakecomputing.com  MyUser  Xpassword123

db  nz  jdbc:Netezza://21.33.14.101:5480/mydatabase  MyUser  Xpassword123
```

The db-alias-name is used when

- using immediate SQL statements to logon and execute 1 statement such as preparing tables for a test,

- using the WORKER statement to define the sessions that will process the queries that have been put into a queue.

You may override the username and optionally the password on the SQL and WORKER statements which makes the DB statement more of a template. Example:

```
sql  tdprod(dbc,dbc)  grant select on TD_METRIC_SVC to acme_benchmark with grant option.

worker heavy nz(acme_hvy01)

worker light nz(acme_lte01:05) 10
```

(On the last 2 statements, all use the password defined in the DB statement. The last statement above says there are 5 usernames from acme_lte01 to acme_lte05 and there will be 10 total concurrent sessions, so each username will be logged on twice. Issue HELP WORKER for details)

BEFORE_RUN – Statements to Execute Before Each Test

This is usually used to insert a row into a TestTracking table on the host DBMS with the RunID, test name, test description and the precise starting timestamp for the execution of a test. By putting the DBMS's current_timestamp into a table, it allows precise extraction of query logs. There may be multiple BEFORE_RUN statements for a TdBench session and all will be executed before each test execution. Syntax:

```
before_run  [“sql”  db-alias-name  …query …  | “os” … os command and parameters | “DELETE” ]
```

Example *(using multi-line syntax with delim=):*

```
before_run sql mydbms delim=eof
insert into acme_benchmark.testtracking
  (TestName, RunTitle, ReportingSeconds, RunId, StartTime)
  values (':testname', ':testdescription', :runsecs, :runid, current_timestamp);
eof
```

Note that if you have database aliases defined for multiple DBMS servers and are running tests on both DBMSs, there is only one set of BEFORE_RUN statements for the current TdBench session so you could be putting the TestTracking information on a different server than the one where the query logs reside.

Also, be careful that the database alias used in the SQL statement is valid because it won't be processed until the run starts.

AFTER_RUN – Statements to Execute After Each Test

This is usually used to update the TestTracking table on the host DBMS with the ending timestamp and other test information after a test execution completes. The combination of the precise starting and ending timestamp of each test makes it easy to get just the rows from the DBMS query logging that were generated during the test. By joining TestTracking to the query logging table, you can use constraints like “where runid = 123” or to compare multiple tests: “where runid in (21,45,77)”. Syntax:

```
after_run  [“sql”  db-alias-name  …query …  | “os” … os command and parameters | “DELETE” ]
```

Example *(using multi-line syntax with delim=):*

```
after_run sql mydbms delim=eof
update acme_benchmark.testtracking set
  ResultCount = :resultcount,
  ErrorCount = :errorcount,
  ActualStopTime = current_timestamp,
  ReportingStopTime = case when reportingseconds = 0 then actualstoptime
timestamp_add(starttime, INTERVAL reportingseconds SECOND) end
where RunId = :runid;
eof
```

AFTER_NOTE – Statement to Execute After the TdBench NOTE command

The TdBench NOTE command will allow you to put text into its internal H2 database TestTracking table documenting the test, such as “good final test” or “failure due to missing table”. If you have dozens or hundreds of tests, this is valuable weeks after the test executes to choose test results for analysis.

The AFTER_NOTE statement allows posting that same notes to the host DBMS. There may only be one AFTER_NOTE statement per TdBench session. Syntax:

```
after_note  run  [“sql”  db-alias-name  …query …  | “os” … os command and parameter]
```

The SQL or OS command **must** contain the variables: :runid and :note. Example:

```
after_note sql mydbms update acme_benchmark.testtracking set runnotes=':note' where runid
= :runid;
```

ZIP – Define Collection of Test Artifacts

OS commands executed during a test (e.g. for ETL) may produce logs with information valuable for analyzing what happened during a test. “Best Practice” is to direct all logs from applications executed during a test to the “temp” directory. The ZIP command to create a single zip file in the Logs directory that is given a name that includes the RunID. Syntax:

```
zip   [work-directory-name]  [logs-directory-name]  {erase}
```

Example:

```
zip temp logs erase
```

TRACE – Capture/Display/Save Console Input/Output

Every input line and output to the console is saved to the trace buffer. When a test is complete, the trace buffer is written to the H2 TestTrackingLog table. Syntax:

```
trace [on | off | list | save {filename} | clear]
```

Hint: Use “trace clear” at the end of tdbench.tdb to clear the setup statement so they don’t end up as the first rows in TestTrackingLog for the first test executed in a TdBench session. .

TdBench 8.01 Tri-Fold

Primary TdBench Commands

# comment
Documents scripts. (Blank lines are ignored).

?
Request current in-progress setup of a test and get suggestions of what else is needed

help [topic | command | ?]
Get details on the syntax and usage of a command

define [test-name] {test description}
Define name and description of a test.

queue [queue-name] [query; | "OS" command; | pathname | wildcard pathname | "kill | stop" {".all"};}
Specify a queue of commands or queries/subqueries for the test. Follow queries with ; or it will be taken as file name.

worker [queue-name] db-alias-name {count} | "OS" {count}
Defines workers to run the queue commands.

run { # ["h"|"m"|"s"] | serial {kill}.}
Starts the test. Example: **run 30m** … executes for 30 minutes

quit
Exits TdBench

Environment/Script Control Commands

cd [directory]
Change TDBench working directory.

class [class-name-of-JDBC-driver] [JDBC Protocol e.g. "jdbc:teradata"] [file-name(s)-of-JDBC-driver-jar]
Load JDBC driver class from a file.

db [alias-name] [JDBC Protocol e.g. "jdbc:teradata://192.168.1.28/database=my_benchmark"] {username {password}}
Define a database alias for use in WORKER and/or SQL commands.

color [input] [output] [error]
sets ANSI screen colors

echo {text:filename | append:filename} {comments + variables} {delim=*string_on_line_later*}
Displays text on console or writes text (with variables) to a file. Can be multi-line with delim=

exec [filename|pathname] {argument-1 ...}
Alias for INCLUDE. Read TDBench commands from a file (can be nested and allows parameters :i1, :i2 …)

include [filename|pathname] {argument-1 ...}
Read TDBench commands from a file (can be nested and allows parameters :i1, :i2 …)

Read [variable] {prompt}
Ask user for value to use in a script

set [variable] {value}
sets or clears contents of a variable

sleep { # {"s" | "m" | "h"} } | {yyyy-mm-dd-}hh:mm{:ss}}
Will go to sleep until a time or a relative time has passed.

status [# seconds | 0]
Will display queue status during test (versus all results.)

trace [on | off | list | save filename | clear]
Controls echo of statements to console, lists out or saves them to file. Example: **trace list** … will list command entered

zip [work-directory-name] [zip-directory-name] {erase}
Defines a directory to archive using zip protocol after test.

Commands to Insert Commands During Test

before_run ["SQL" ... sql command; | "OS" ...os command; | "DELETE"]
Statements to be run before RUN statement. (allows substitution of TdBench variables)

after_run ["SQL" ... sql command; | "OS" ...os command; | "DELETE"]
Statements to be run after the test completes

before_worker [queue-name] [query; | "OS" command; | pathname]
Specify one or more queries to run in each worker before a queue. (e.g. database command)

before_query [queue-name] [query;]
Specify a query or command to run before each subquery in a queue. (e.g. Query_Band)

after_query [queue-name] [query;]
Specify a query or command to run after each subquery in a queue.

after_sql [tdbench-command | "delete"]
Specify command(s) to execute to evaluate a SQL :retcode or :linecount

after_os [tdbench-command | "delete"]
Specify command(s) to execute to evaluate an OS :retcode

Test Control Commands

at [queue-name] [#]{"s" | "m" | "h"}
Time or relative time to start a queue.

Debug [on | off]
Suppress RUN, OS, and SQL during testing

explain [queue-name]
Adds "explain" to every subquery in a queue.

finish [queue-name]
Ensures queue must finish at least once before the test ends.

limit [queue-name] [#]{"s" | "m" | "h"}
Maximum time to run a subquery in a queue before aborting.

pace [queue-name] [#]{"s" | "m" | "h"}
interval between queries in a queue.

param [queue-name] [file delimited parameters] {delimiter | tab} {count}
Provides parameters for subqueries in a queue.

prepare [queue-name] {data-type-1, ...}
Provides parameters for subqueries in a queue run using prepared statements.

rowcount [queue-name]
Workers in a queue will count all returned rows (affects performance!)

rows [queue-name] {maximum row count}
Workers in a queue will returns some or all rows (can affect performance!)

Queue Preparation Commands

replicate [queue-name] [multiplier]
Duplicates a queue's commands by a multiplicative factor.

save [queue-name | runid] [pathname-template-to-save-into]
Saves a queue's commands in a set of files for future tests or the statement in a runid into a file.

shuffle [queue-name]
Reorders a queue's commands and parameters randomly.

Immediate Commands

goto [labelname | testname]
Skips until LABEL or DEFINE statement with the label/test name specified

if [variable] [= | != | > | >= | < | <=] [constant] ["THEN" statement]
Tests results of prior OS, SQL or RUNs to determine next statement, test name or label to process

label [labelname]
Provides a name to be referenced by GOTO

OS { file:fileame | file:null } [command-to-execute]
Immediately executes an operating system command

sql { file:filename | file:null | text:filename} [db-alias-name] [sql-command";" | filename-of-sql-commands]
Immediately executes one or more SQL statement

Variables:

Type	Examples
INCLUDE or EXEC put in file being run	:i1 :i2 :i3 …
Environment May be different in Windows or Linux. Use SET command in Windows or export in Linux to set	${HOSTNAME} ${LOGNAME) ${PWD} ${USERNAME} ${MyPass} ${scale}
Program Variables Use in script during or after test. Use TdBench echo :all for complete list	:runid :testname :runsecs :resultcount :linecnt :retcode
PARAM variables from each delimited field in PARAM file	:1 :2 :3 …
Worker Variables Use in queries, substituted dynamically during execution	:runid :testname :queryname :queue :timestamp

H2 (Results) Database Commands

archive [tables-prefix-name] {"from" other_prefix} {"where" where-conditions]

Archives tests in H2 in separate tables. Optionally may copy from another archive and apply where…

delete [runid]

Removes rows from TestResults table. Useful when test fails with thousands of errors

Dump {archive-name} {"to" zip-prefix}

Writes all test information from H2 archive to CSV files and zips as archive-name.zip or zip-prefix.zip.

list {archive-name} {-count | -recent-count | runid {to runid] | "where" where-conditions}

Lists all tests captured in the H2 database.

note [runid] {add} {notes or observations to set, add, or omit to delete note}

Adds a note to a RunID describing conditions or usefulness of test to final results

remove [tables-prefix-name]

Deletes archives data in H2

Restore [archive-prefix] {"to" other-archive-prefix}

Restores tests archived/dumped from a zip file to tables with archive-prefix or other-archive-prefix

select [h2 SQL select query]

Allows query against test results output to console (alias for SQL H2 SELECT...).

H2 Tables

TESTTRACKING ***… the version inside TdBench's H2 database***

RUNID
TESTNAME
TESTDESCRIPTION
STARTTIME
STOPTIME
LOGCOUNT
RESULTCOUNT
ERRORCOUNT
NOTES
URL

TESTTRACKINGLOG table in H2

RUNID
LOGLINENO
LOGTIMESTAMP
LOGTYPE … Input, Info, or Error
LOGTEXT

TESTRESULTS

RUNID
QUERYID
SUBQUERYID
QUEUENAME
WORKERID
USERNAME
QUERYNAME
RETURNCODE
ERRORMSG
STARTTIMESTAMP
RESPTIMESTAMP
RESPTIMEMS
ROWS
DEFICITMS
PARAMS

H2 Analysis Views

RPT_TESTS

Summarizes at the RunID level.

RUNID	*group by*
TESTNAME	*group by*
STARTTIME	*group by*
STOPTIME	*group by*
RUNSECS	*group by*
NUMEXEC	*calculated*
NUMNOERROR	*calculated*
NUMQUEUES	*calculated*
NUMWORKER	*calculated*
AVERESP	*calculated*

RPT_QUEUES

Summarizes by queue within each RunID

RUNID	*group by*
TESTNAME	*group by*
QUEUENAME	*group by*
NUMWORKER	*calculated*
NUMEXEC	*calculated*
NUMNOERROR	*calculated*
AVERESP	*calculated*
MINRESP	*calculated*
MAXRESP	*calculated*

RPT_QUERIES

Summarizes by query name within queue within each RunID.

RUNID	*group by*
TESTNAME	*group by*
QUEUENAME	*group by*
QUERYNAME	*group by*
NUMQUERY	*calculated*
NUMEXEC	*calculated*
NUMNOERROR	*calculated*
AVERESP	*calculated*
MINRESP	*calculated*
MAXRESP	*calculated*

RPT ERRORS

Summarizes errors by RunID, queue, query

RUNID	*group by*
TESTNAME	*group by*
QUEUENAME	*group by*
QUERYNAME	*group by*
RETURNCODE	*group by*
ERRORMSG	*group by*
NUMERRORS	*calculated*