TrueBench

Simple setup. Realistic Workloads. Efficient analysis.

Date: January 20, 2026

Release: truebench_01_00

TdBench 9.01 Reference Manual

INTRODUCTION – TrueBench Overview

Purpose

TrueBench is an interactive and script-driven framework for evaluating how database and system workloads behave under change.

Rather than managing ad-hoc test scripts, temporary logs, and shell loops, TrueBench provides a repeatable and organized way to define, execute, and analyze realistic workloads across different platforms, configurations, software releases, or workload compositions.

TrueBench is designed for enterprise architects, DBAs, application developers, and database engineers who need to understand the impact of change on production behavior— whether that change involves new hardware, new software, new parameters, new indexes, or new applications.

Why Use TrueBench

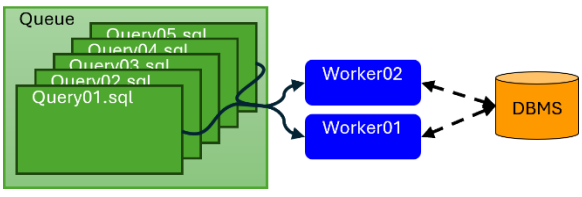
- **Fast to start:** Point it at your SQL directories — no complex setup or custom coding.
- **Scalable:** Run tests in one session or hundreds of concurrent threads.
- **Repeatable:** Each test is defined, logged, and tracked in the metadata database for comparison.
- **Flexible:** Supports SQL and OS commands, parameter substitution, conditional logic, and pacing controls.
- **Portable:** Works with multiple DBMS drivers via JDBC or Python connectors.

In short: **one consistent, scriptable environment** replaces dozens of local scripts, log files, and ad-hoc experiments.

Core Concepts

TrueBench centers around two main ideas: `Queues` and `Workers`.

- A **queue** is a list of SQL files or OS commands that make up a unit of workload. There may be multiple queues.



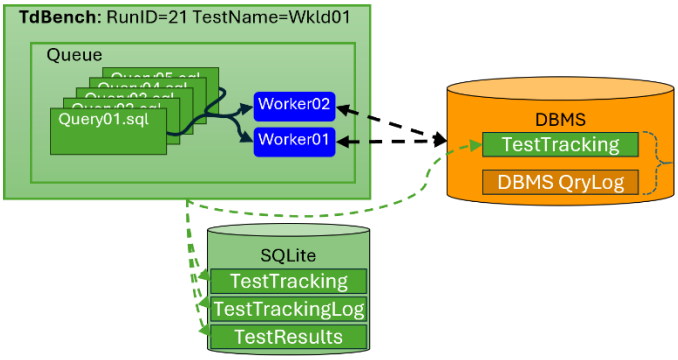
- A **worker** is a session (or multiple sessions) assigned to run the contents of a queue against a database alias.

A test may include any number of queues and workers. Simple tests may run a single queue once, while more advanced tests may involve multiple queues and multiple workers running concurrently.

Tests may **run** as
- **fixed work**, where every statement is executed once with the defined concurrency,
- **serial**, where each statement in the test is executed by itself, useful for query validation, or
- **fixed time**, where TrueBench maintains a constant workload for a specified duration.

The **fixed time** test is useful for simulating production behavior. Queues can be established with special queries or OS commands set to fire at a time relative to the start of the test to simulate stress conditions or adverse events. Queues can be created with a list of queries, each having a start time associated that can be used for query replay. Query replay is useful for validating that a platform can meet requirements, but is not useful for measuring DBMS capacity.

Every command executed during a test — including the setup commands, SQL, runtime messages, and error diagnostics — is stored in the **metadata** database which is implemented with **SQLite**. Each execution receives a unique `runid`. All timings, session statistics, return codes, and error details are tracked for later review and comparison.

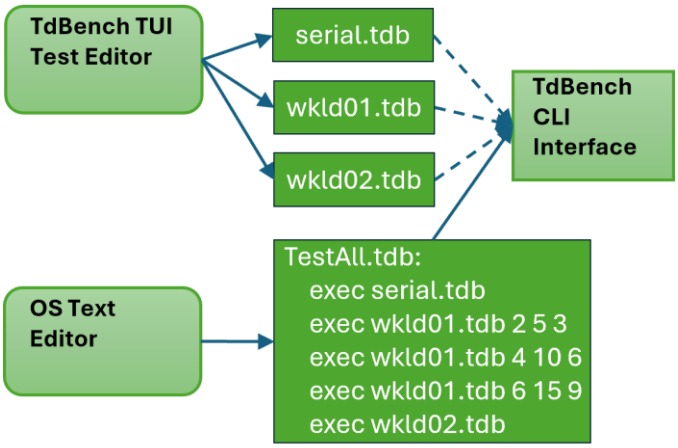


TrueBench can coordinate test tracking with a host DBMS TestTracking table so the precise starttime and stoptime for each runid/testname can be used to join with host DBMS query logging capabilities to understand how each query in the test utilized host DMBS resources and the impact of workload

Using TrueBench: CLI and TUI

TrueBench can be used entirely from the command line or through the integrated TUI test editor.

- The **cli** interface is ideal for scripting, automation, or



interactive execution.

- The **tui** interface provides a structured editor for building test definitions, browsing help topics, and interactively editing queues and workers.

Both interfaces share the same scripting language and metadata database, and tests created in the TUI can be executed from the CLI — and vice-versa.

Suggested Reading

Any of the following help topics can be opened with the TUI F1 key, with `HELP <topic>` for text output or with the `MAN <topic>` for a full screen help browser:

- **cli** — scripting basics, commands, examples
- **tui** — using the text-based editor and navigation features
- **metadata** — structure of the metadata database and examples
- **sqlite** — SQL examples for inspecting results and logs
- **setup** - Overview of setting up to run TrueBench
- **db** — defining database aliases and drivers
- [parameters] — SQL parameters and parameter files
- **queue** — how queues are built and used
- **worker** — how workers execute queues
- [[programming]] — IF, GOTO, DO/END, READ, and SET

For a full list of topics or commands:

- [[_all_topics]]
- [[_all_commands]]

Overview Of Manual

TrueBench Commands

Getting help

The CLI will give a short help with syntax only if you follow the command name with ?. You may get additional help using the `HELP` command or the `F1` key when running the TUI screen editor with the `SCREEN` command.
help - Display command information and topics
man - Full screen viewer of command information and topics
screen - Open the full screen editor for simple tests

Initial setup for TrueBench

To reference a database you must define a database alias with at least the URL, username and password. You can use the `VALIDATE` command to test it, then put the DB statement (and optionally other startup commands) into the truebench.tdb file.
db - Define database alias for SQL and Worker commands
validate - Test the database alias and measure latency

Basic commands for any test

You can either enter these statements interactively, put them in a file, or generate them with the `SCREEN` command. **define** - Start a new test
queue - Define set(s) of queries and commands for the test
worker - Assigns execution threads to queues
run - Executes the test with optional duration
exit - Terminate TrueBench. `quit' is an alias

Advanced commands within a test definition

These commands must follow the `QUEUE` command that defines the queue they modify. The `PARAM`, `PACE`, `ROWCOUNT`, and `ROWS` commands can be input and edited under the `SCREEN` command.
param - Defines query parameters to substitute for :1, :2, ...
[pace] - Control the rate or % execution for queue(s)
rowcount - For initial query validation, return and count all rows
rows - For initial query validation, display some # of rows
limit - Limit how long any query can run (some DBMSs)
before_worker - Define query to execute by each worker before test starts
before_query - Define query to execute before each query in the queue
after_query - Define query to execute after each query in the queue

Immediate Commands

These commands are typically placed before the `DEFINE` statement, but if encountered within a test, are executed immediately. Within a test, the `QUEUE` command defines SQL

and OS commands to execute by worker(s).
sql - Immediately execute a SQL query on an DB alias
before_sql - Define statement to execute prior to subsequent SQL statements
os - Immediately execute an OS command
cd - Change the current directory or reset to “home”

Leveraging the Metadata from tests

The following commands use and modify the metadata captured from test executions. You can also use the alias “me” on SQL statement for a larger set of manipulations. Additional information may be found in the topics: **metadata** and **sqlite**.
list - List stats on tests, the SQL in a test, or defined aliases
select - Alias of SELECT using the “me” alias for metadata tables
note - Apply a note to a completed test to document conditions, issues or result

Reusing code saved in files

The INCLUDE and EXEC are aliases of the same functionality. Lines are read from the referenced file. Words that follow the file name will be taken as arguments to substitute into :i1, :i2, etc.
include - Read lines from another file, may be nested
exec - Alias of include

Linkage to other Databases

The system variables set about the defined run or the results of the prior run are available to use in before/after specifications. “Before”, you might insert a rows into a testtracking table on the host DBMS and “After”, you might update that runid with the ending time and other statistics. By maintaining a host DBMS testtracking table, you can join it to DBMS logging tables with its timestamps to analyze what the DBMS was doing that impacted test results. These statement are often placed in the truebench.tdb startup file.
before_run - Statements to execute before workers start running
after_run - Statements to execute after run completes
after_note - Allows replicating a note to a database or file

Scripting Capabilities

TrueBench supports basic scripting capabilities to automate streams of tests. These statements are supported by the CLI and can be used to “glue” together multiple tests created by the full screen test editor. An overview may be found in the [[programming]] topic.
echo - Puts text to the screen or file, with subsitution of variables
set - Defines or unsets a variable
read - Sets a variable with a prompt
if - Simple test between 2 variables/literals to execute another command
else - Defines statement(s) to execute if last IF statement was false
goto - Allows skipping within a file to a LABEL or testname on a DEFINE

label - Defines a label within a file that can be referenced by `GOTO`
do - Defines a block of code to execute until the `END` statement
trace - Control output detail as statements are executed for debugging
profile - Allows changing the default directories and other settings.

All Topics

Use the up/down arrows to select a topic, then press enter key to open that topic

Basics:

introduction - Overview of TrueBench
tui - Text User Interface overview for the Test Editor screens
cli - Command Line Interface Overview
setup - How to configure database aliases
[[drivers]] - Overview of using DBMS drivers to connect to your data source

TrueBench Screen Editor

[[tui_main]] - Description of the Main Test Editor Screen
[[tui_queue]] - The Queue Editor Screen defines a set of work and workers to execute it
[[tui_help]] - Description of the full screen help viewer
[[tui_run_monitor]] - Monitors progress of test execution

Analyzing the data captured from each test

metadata - Description of tables describing each test
sqlite - Basic Syntax supported in SQLite used for Metadata

Advanced Programming

[[variables]] - How variables can generalize scripts
[[programming]] - scripting commands to automate a series of tests

Architectural information on the Python implementation of TrueBench

[[cli_architecture]] - How CLI modules are organized
[[tui_architecture]] - How TUI modules are organized

Different modes of TdBench Usage

TUI – Text User Interface Overview

The TrueBench TUI is designed for creating basic tests quickly and interactively. The editor generates the core test statements: `DEFINE`, `QUEUE`, `PARAM`, `ROWCOUNT`, `ROWS`, and `RUN`. These can be executed immediately or saved to a file for later use.

Multiple tests may be created with the TUI, then combined or extended using the CLI, where advanced scripting (`IF`, `GOTO`, `READ`, `SET`, `DO`/`END`) and complex queue options are available. Advanced `QUEUE` features not supported by the TUI will be flagged as errors if loaded.

Why a TUI?

The full-screen editor is implemented as a Text User Interface to provide excellent responsiveness when running TrueBench on a server near the database. A GUI would require an XWindows client and transmit high-volume mouse and screen updates across cloud latency, making it slow or unusable in many environments. The TUI avoids these issues while remaining fully interactive.

Field Types and Navigation

- Entry fields appear as `[_____]`.
- Dropdown selections appear as `{ option }`.
- Use the arrow keys to move between fields.
- `Tab` and `Shift+Tab` switch between logical sections on the screen.
- Function keys perform actions such as saving, running, browsing files, or returning to the previous screen (exact bindings vary by screen).

TUI Screens

- **Main Screen**
  Defines the test name, description, queues, and run options (Fixed Work, Fixed Time, Serial).
- **Queue Editor**
  Adds SQL or OS work items, file references, parameters, workers, pacing, RowCount, and sample row output.
- ***File Picker***
  Provides a basic file browser used by the **Main Screen** to open or save files and use

by the **Queue Editor** to identify seach patterns for files containing queries or parameters.

- **Worker Execution (Run View)**
  Displays real-time test activity, including worker status, errors, and summary results.

More Reading

[[TUI_MAIN]] – Test Editor Main Screen
[[TUI_QUEUE]] – Queue Editor
[[CLI]] – Full command language
[[INTRODUCTION]] – General overview of TrueBench
[[METADATA]] – Understanding test history and results

CLI – Command-Line Interface

Commands in TrueBench are case-insensitive and can be issued interactively or placed into `.tdb` script files. - Simple tests can be defined with just four core commands — **define**, **queue**, **worker**, and **run** — making it easy to build and execute basic workloads. - More advanced workloads can use parameterized SQL, multiple queues, or INCLUDE scripts to reuse components across tests. - Complex multi-stream workloads and replay scenarios can be composed using the extended scripting language: **set**, **read**, **if**, **goto**, and **do**/[[end]].

Quick Start

1. Place your SQL files in a directory under the TrueBench installation, for example:

```
myqueries/
```

Each file may contain one or more SQL statements. By default, each file is queued as a single SQL script.

2. Define a database alias (see **db**):

Example:

```
db mydb teradatasql dev.mycompany.com user=myuser password=mypwd
```

3. Create a simple test interactively or in a `.tdb` file:

```
define firsttest this is my first test of TrueBench
queue q1 myqueries/*.sql
worker q1 mydb
run
```

The test executes all queries once in a single session and logs results to the metadata DB.

4. To run with five concurrent sessions:

```
worker q1 mydb 5
```

5. To run continuously for 15 minutes, use one of the following equivalent options:

```
run 15m
run .25h
run 900s
```

Multiple Workload Queues

You may organize queries by runtime or purpose and simulate production variation. Since there are more lines, you may want to save in a file such as workload01.tdb.

```
define prodworkload Production simulation for Call Center, Sales, and
Marketing

queue callctr    callctr/*.sql
param callctr    callctr/*.param
worker callctr   mydb 3

queue sales      sales/*.sql
param sales      sales/*.param
worker sales     mydb 5

queue marketing  marketing/*.sql
worker marketing mydb 1

run 15m
```

To execute the script:
`exec workload01.tdb`

To restrict frequency of fast queries, add after `queue callctr...`:
`pace callctr 15%`

Parameterized Queries

Example SQL:
`where region = ':1' and extract(month from order_date) = :2`

Example `.param` file:
```
east|03
west|03
south|01
```

Each line represents a separate parameter set. The root of the filename of the parameter file must match the root of the query file name. Example, `callctr/query01.sql` matches `callctr/query01.param`. Then add the statement after the queue is defined:
`param callctr/*.param`

Coding Notes

1. Commands are case-insensitive: `DEFINE`, `define`, and `Define` all work.

2. Tests must begin with **define** and end with **run**.

3. Commands that reference the queue name must follow the **queue** statement.

4. Immediate commands such as **help**, **os**, and **sql** run immediately, even inside test definitions.

Getting Help

If a ? is passed to any commend you will get a short summary of syntax and purpose:
`> define ?`

Result:

DEFINE description
Start a new test run and record its definition in the tracking database.

Help topics: - [[_all_topics]] — list of all help topics
- [[_all_commands]] — list of all commands grouped into categories

Listing of known drivers and defined database aliases: - `list drivers` - show the definition of known database drivers to use on the **db** command - `list db` - show the list of currently defined database aliases (via the **db** command)

Startup Behavior

On startup TrueBench opens:

`testtracking.db`

If the writer lock is held by another session, TrueBench enters read-only mode.
You may still issue `list`, read-only **sql**, and browse documentation.

Automatic Include

If `truebench.tdb` exists, it runs automatically.

Example (prompted variables):

```
set user = johnsmith
read password Enter the password for this session:
db mydb teradatasql dev.mycompany.com user=:user password=:password
```

Example (referencing previously set environment variables):

```
db mydb teradatasql dev.mycompany.com user=${user} password=${password}
```

Exiting

Use **exit** or `quit` to terminate TrueBench.

On exit:

- Worker threads stop
- Metadata commits are finalized
- Locks are released
- DB connections close

Installation and Setup

SETUP – Preparing TrueBench for Use

Before using the TUI or CLI, TrueBench must know how to connect to your database systems. This is done by creating a `truebench.tdb` file in the TrueBench home directory. The file may contain one or more **DB** commands that define database aliases. Aliases allow tests to run against different systems or with different logon identities. This startup file may contain additional statements you want executed on every startup.

Examples of why multiple aliases are useful: - Running queues with different users for OS workload management
- Assigning administrative queries to a privileged ID
- Using end-user credentials having different grants for table access

Passwords may be supplied several ways:
- `?` to prompt the user when the DB command executes
- **READ** statements to request credentials interactively
- External wrapper scripts that retrieve passwords from a secure store

A DB alias is defined with: `DB <alias> <driver> <url> user=<id> password=<value>`

Once you have defined an alias, you may use it in various commands. For example, if you defined an alias of `td` for Teradata, you could issue:

```
sql td select current_timestamp
```

or:

```
worker q1 td 5
```

to start 5 sessions executing queries against Teradata.

Verify Connectivity

Before using TrueBench, verify that the client platform can reach your DBMS. Use a tool you already know (CLI, BTEQ, isql, SQL*Plus, or driver-specific utilities) to confirm that:
- firewall rules are correct
- hostnames and URLs resolve properly
- credentials are valid

This avoids diagnosing network issues inside TrueBench.

Validate with TrueBench

After creating `truebench.tdb`, start TrueBench and issue: `VALIDATE <alias>`

The **validate** command checks that the driver loads, the session opens successfully, and performs a short test to validate best possible performance with your network and client platform.

Using SCREEN Automatically

If most of your work involves simple test creation and execution,
use the **SCREEN** command
as the last line of `truebench.tdb`. TrueBench will open the TUI test editor automatically after loading your DB aliases.

You may also invoke `SCREEN` interactively from the CLI at any time.

Examples

```
DB td teradatasql dev.mycompany.com user=benchmark password=Pas$word146
```
This defines an alias with an explicit password. Be careful of the security risk when TrueBench is used on a platform accessed by multiple people.

```
DB admin teradatasql dev.mycompany.com user=dbc password=?
```
This alias that prompts for the password every time it is executed ... (when TrueBench starts)

```
read pw Enter the password for all users:
DB sales teradatasql dev.mycompany.com user=bench_sales password=:pw
DB mkt  teradatasql dev.mycompany.com user=bench_mkt password=:pw
```
Prompted password for the users, each time truebench.tdb is executed.

```
DB sales teradatasql dev.mycompany.com user=bench_sales password=${BenchPW}
DB mkt  teradatasql dev.mycompany.com user=bench_mkt password=${BenchPW}
```
Leverages a password set in an environment variable before TrueBench starts. The `set` command can be used in Windows scripts, but you must use `export` command in Linux to have the variable available when TrueBench runs.

Additional Reading:

`help db` - Syntax of the `DB` command to define database aliases
`help validate` - Usage of Validate to check connection and test latency
`help profile` - change standard directory usage and settings

DB — Define a database connection alias

Syntax:

```
DB <alias> <driver> url=<url> [user=<user>] [password=<pwd | ?>]
[param=value ...]
```
Defines a named connection alias using one of the supported drivers.

Description:

The DB command registers a database connection alias used by **SQL** and **WORKER** commands. [[Drivers]] determine how connections to the DBMS are established. If the driver module is missing, TrueBench can attempt to install known drivers dynamically. Any

DBMS or data source can be accessed if you set up its [[ODBC]] driver manually. The **LIST** command can show you known drivers and installed [[ODBC]] drivers.

Most DBMS drivers require additional connection parameters beyond URL, user, and password. These parameters are driver-specific and are passed unchanged into the underlying Python or ODBC connector.

Examples of common driver-specific parameters include: - database / dbname - port - schema - warehouse - role - ssl / tls settings - authentication method / logon mechanism - driver-specific timeouts

See `help <driver>` or `man <driver>` for the parameter list and examples for each driver.

Parameters:

- `alias` — Short name for referencing this connection later.
- `driver` — One of the supported driver keys (sqlite3, teradata, snowflake, odbc, etc.).
- `url` — Connection URL or path (typically a hostname, server identifier, or database file).
- `user` — Optional username.
- `password` — Password or `?` to prompt securely.
- `param=value` — Additional driver-specific parameters.

Special parameters commonly required by some drivers: - `database=<dbname>` — Required by some DBMS platforms (or specific connection styles). - `port=<port>` — Required if the DBMS is not using its default listener port. - `driver="<odbc driver name>"` — Required for the generic `odbc` driver. - `module=<python module>` — Required for the generic `pydbapi` driver.

Examples:

- `DB local sqlite3 url=testtracking.db`
  Defines a local SQLite database for logging and metadata.

- `DB td teradata url=myvantage.company.com user=dadmin password=? encryptdata=true`
  Prompts for password and defines a Teradata Vantage connection (Python driver).

- `DB myodbc odbc url=myserver user=myuser password=? driver="ODBC Driver 17 for SQL Server" database=mydb`
  Defines a generic ODBC connection using an installed OS ODBC driver.

- `DB pg pydbapi url=localhost user=myuser password=? module=psycopg2 database=mydb port=5432`
  Uses the generic Python DB-API driver (requires the module to already be installed in `.venv`).

Notes:

- Use `LIST DRIVERS` to see the drivers supported by this TrueBench release.
- Use `LIST DRIVER <driver>` to show driver metadata and installed status.
- Use `LIST DB` to list currently defined DB aliases.
- Use `LIST ODBC` to list installed OS ODBC drivers (requires `pyodbc`).
- After defining an alias, run `VALIDATE <alias>` to confirm connectivity and baseline latency.

VALIDATE — Test a database alias for latency and tactical performance

Syntax:

`VALIDATE <alias> [max_seconds]`
Tests connectivity and basic performance of a database alias defined using the DB command.

Description:

The VALIDATE command performs two checks:

1. **Connection Test**
   Attempts to connect using the DB alias (URL, driver, user, password).
   Reports the time required to establish a session.
2. **Trivial Query Loop**
   Executes a simple SQL statement: `SELECT 1`

The loop runs either: - up to **100 queries**, or
- for **max_seconds** (default = 5 seconds),
whichever comes first.

For each iteration, the response time is measured, and after completion: - Number of queries executed - Average response time (ms) - Minimum and maximum response times (ms) - Queries per minute (QPM)

are reported.

This is not a benchmark of DBMS computational power; it is primarily a test of **network latency**, **driver overhead**, and **client-side responsiveness**.

Parameters:

- **alias** — Database alias previously defined with the DB command.

- **max_seconds** — (Optional) How long to run the trivial loop. Must be numeric. Default = `5.0`.

Examples:

- `VALIDATE td1`

- `VALIDATE local_db 10`
  Runs trivial SQL on alias `local_db` for up to 10 seconds.

What do timings mean?

When TrueBench runs on your laptop but the DBMS resides in the cloud, network latency becomes the dominant component of tactical query response time.

The response measurements from VALIDATE include:

1. Time for the request to reach the DBMS
2. Time for the DBMS to execute the trivial query
3. Time for the result to return
4. Driver, Python, and OS overhead

Even if the DBMS itself executes a trivial query in **1 ms**, a cloud round-trip can add **40–80 ms** or more.
Therefore:

- **You cannot measure faster tactical response times than your network latency allows.**
  The "average response ms" reported by VALIDATE becomes the *lower bound* of any tactical query you can measure from that client.
- Latency has **little impact** on long-running or multi-second queries.
- For tactical (short) queries, latency can fully mask DBMS performance.
  If DBMS A returns in 1 ms and DBMS B in 5 ms, both may appear identical (e.g., ~60 ms total) when tested over a 60 ms round-trip network.

If comparing multiple DBMS systems — or running tactical workloads — you must ensure the **same network conditions** for each system.

Best Practice:

Run TrueBench from a VM **in the same cloud region and availability zone** as the DBMS you are testing.

This minimizes network overhead and shows a clearer picture of the true DBMS execution speed.

Notes:

- This command is safe and non-destructive.
- It is intended primarily as an environment sanity check before running full tests.
- Errors reported by VALIDATE usually indicate connection issues, authentication problems, firewall restrictions, or driver misconfiguration.

Help and the TUI Manual

There are two interfaces for getting help on using TdBench: - F1 help key in the TUI interface - The CLI **MAN** command which displays the help in a TUI window - The CLI **HELP** command which streams help text to the console

In the CLI interface, you can also pass the ? as the arguement to any command to get the syntax and a 1 line description of that command.

TUI Help Viewer

The TrueBench help system can be viewed in two ways:

1. CLI text output using the **help** command
2. Full-screen TUI viewer using the **man** command or by pressing **F1** inside the TUI

The TUI (Text User Interface) viewer displays help topics in a scrollable full-screen window. It does not use a mouse; navigation is performed entirely with the keyboard.

Interface Basics

The viewer shows formatted text with headings, code fragments, and clickable help links such as:

introduction

Use the arrow keys to move between visible lines.
If a line contains one or more help links, the left/right arrows move the active highlight between them.
Press **Enter** to open the highlighted link.

Navigation Keys

Up / Down
Move the selection cursor through the displayed lines

Left / Right
When a line has one or more links, move between them

Enter
Open the highlighted link or follow the selected reference

Esc
Close the viewer and return to the previous screen

PageUp / PageDown
Scroll by a full page at a time

Home / End
Jump to the beginning or end of the help document

F1
Open this TUI help topic from anywhere inside the TUI

F2
Open the file browser (used in forms and editor modes)

F3
Exit the current TUI mode and return to the prior menu

F5
Show the list of help topics

F6
Show the list of TrueBench commands

Using Links

Links appear in bold with an underline such as **introduction**.
To activate a link:

1. Move the cursor to the line containing the link using Up/Down
2. Use Left/Right to highlight the exact link
3. Press Enter to open it in the viewer

This allows navigating quickly through related help topics without returning to the CLI.

Opening Help Files

From the CLI:

`man <topic>`
Opens a help file in the TUI viewer

`help <topic>`
Shows the same topic directly in the console

From within the TUI:

Press **F1** to open help for the current screen
Press **F5** to browse all help topics
Press **F6** to browse all TrueBench commands

Notes

The TUI help viewer supports a limited Markdown subset:

- # Headings
- *Subheadings*
- `inline code`
- **bold** and *italic*
- ***bold italic***
- Help links in the form [[topic]]

Help files are located under the `help/` directory and use the following extensions:

- `.topic` for general topics
- `.command` for CLI commands
- `.driver` for database driver documentation

MAN – View Help Topics in the TUI Help Viewer

Syntax:

```
MAN {topic}
```

Description:

Open a help topic using QueryDriver's full-screen Text User Interface (TUI) viewer. This viewer supports keyboard navigation: the Up/Down arrows move the highlight between

lines, and when a line contains one or more links, Left/Right arrows cycle among them. Press Enter to open the highlighted link and jump directly to the corresponding help file.

The MAN command is similar to the CLI `HELP` command, but offers: - smooth scrolling for long documents
- keyboard-driven navigation
- ability to activate cross-references by highlighting them and pressing Enter
- clearer formatting for multi-section help files

If the topic does not exist, MAN reports an error.

Parameters:

- **topic** – Optional. If omitted, the **introduction** is displayed.

Examples:

- `MAN introduction`
  View the introductory help file in the full-screen viewer.

- `MAN _all_commands`
  View the [[_all_commands]] index of all TrueBench commands

- `MAN _all_topics`
  View the [[_all_topics]] index of all TrueBench topics

For details on navigation keys and link activation, see [[tui_help]].

HELP — Display usage information about a command or topic

Syntax:

HELP { command | topic | ? } Displays help text for a command or topic.

If used without arguments, shows the **introduction** topic.
If the argument is **?**, displays this short syntax and one-line description.

Parameters:

- **command** — The name of a command (e.g., DEFINE, QUEUE, RUN).

- **topic** — A general documentation topic (e.g., VARIABLES, INTRODUCTION).

- **?** — Prints a brief syntax and purpose summary for the HELP command itself.

Examples:

- `HELP` — Displays the *introduction.topic* file.

- `HELP DEFINE` — Shows full help for the DEFINE command.

- `HELP INDEX` — Lists all available commands and topics.
- `HELP ?` — Shows short syntax and description for HELP itself.

Using the Text User Interface Test Editor

SCREEN – Full-Screen Test Editor

Syntax:

```
SCREEN
```

Description:

The **SCREEN** command launches the interactive full-screen editor for simple tests that allows you to define:

- testname and description
- queues and their SQL/OS commands
- worker assignments
- parameters (`PARAM`), pacing (`PACE`), row display (`ROWS`), rowcount mode, etc.

The editor provides structured fields and screens to guide test creation without requiring manual `.tdb` scripting.

Press **F1** anywhere in the editor to bring up context-sensitive help.

Behavior:

- Opens a curses-based full-screen environment
- Edits one test at a time
- Produces a `.tdb` script identical to what a user would type manually
- Press **F5** to execute the test using the CLI engine
- Saving, loading, and file management are available from the editor menus

All SQL execution, worker orchestration, and metadata logging remain under the authority of the CLI engine. The TUI editor only constructs the test script.

Navigation:

- Arrow keys to move between fields
- Enter to edit fields
- Function keys for major actions (F1 Help, F5 Run, F6 Save, etc.)

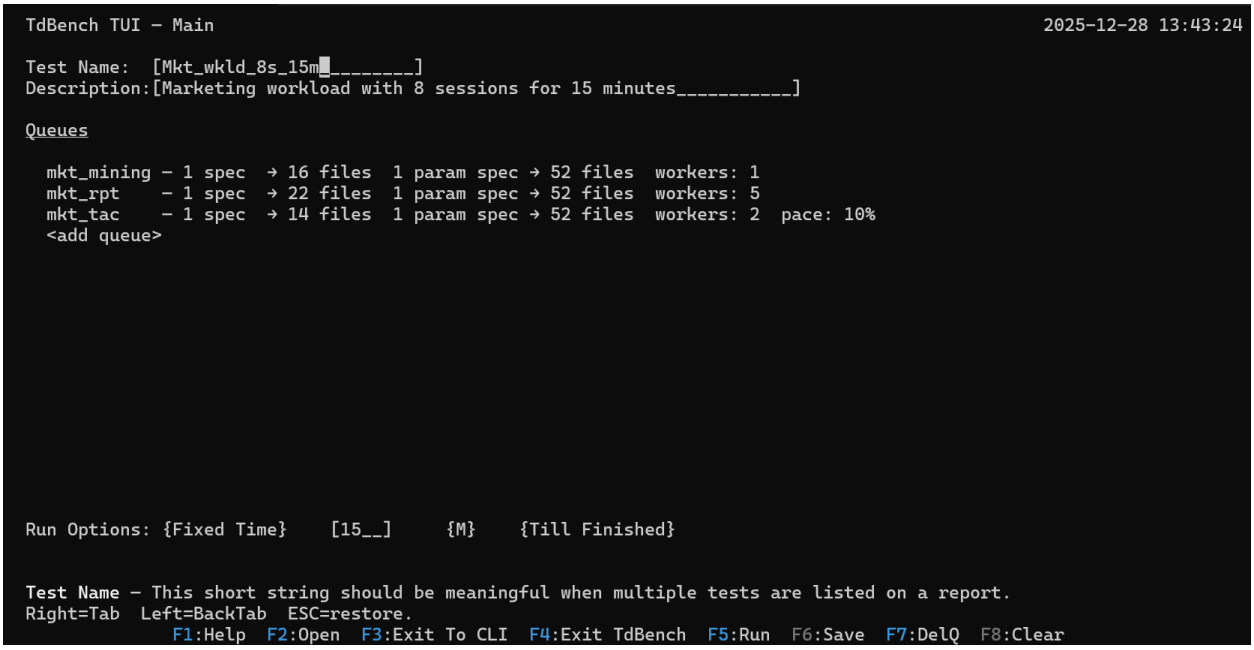
See [[tui_help]] for detailed navigation instructions.

Examples:

- SCREEN
  Opens the full-screen editor where you can visually define and run a test.

TUI MAIN SCREEN – Create or Update a Test Definition

The Main Screen defines a single test: its name, description, queues, run mode, and



optional timing. It is invoked using the **SCREEN** command.

You can navigate through the screen using arrow or tab keys. Entry fields appear as [_____] and dropdown selectors appear as { option }.

A test consists of one or more queues, each with defined work items and worker counts. Each queue could contain a set of queries used by different organizations sharing the DBMS or could be categorized by the kind of response time required.

The most realistic simulation of production requirements can be achieved by breaking up the workload into queues and varying the concurrency and pacing of each queue to mimic the workload of the production environment. Then set the test to run for a fixed period of time to maintain a steady level of workload. It is also possible to create a queue with unusual events to occur during the test or initiate OS commands to trigger external events. A queue can even be given a list of queries and related times relative to the start of the test for query replay.

Test Name and Description

Use [_____] fields to enter a short test name and a longer description. Consistent test naming helps organize larger test suites. When tests complete, results of the last 3 runs of the same test name are listed for comparison.

Queues

Each queue defines a workload: a set of work items and a worker count. A test must contain at least one queue but may include many. Select a queue and press `Enter` to edit it. Use `<add queue>` to create a new one. That will bring up the [[TUI_QUEUE]] screen

Run Options

The Run Options line determines how the test is executed. The mode is selected from a dropdown `{ Fixed Work | Fixed Time | Serial }`.

When `{ Fixed Time }` is selected, two additional fields appear:

- [____] – Duration value
- `{ s | m | h }` – Units: seconds, minutes, hours

These fields are *only shown* in Fixed Time mode. For Fixed Work and Serial, the duration and unit fields are hidden.

A final field selects the completion policy:

- `{ Till Finished | Kill at end of time }`

`Till Finished` allows any active query to complete after time expires. `Kill at end of time` requests that the worker terminate running queries immediately when the timer ends. Not all DBMS drivers support an immediate kill; many will wait for the query to respond to cancellation.

Fixed Work
Workers run all queue items once (expanded by parameters). When a worker finishes early, concurrency decreases naturally. Useful for correctness and content validation.

Fixed Time
Maintains full concurrency for the entire duration. Workers continue executing tasks until the timer expires. Best for simulating steady production load.

Serial
Runs each queue one at a time with a single worker. Produces the clearest measurement of individual query performance. Often used with `RowCount` or `Rows` options (in the Queue Editor) during validation of the queries in the benchmark.

Key Usage

Up/Down – move between fields
Tab / Shift+Tab – switch columns/fields
F1 – Help
F2 – Open an existing test definition (DEFINE...RUN only; no scripting)
F3 – Exit to the CLI
F4 – Exit TrueBench
F5 – Run the current test
F6 – Save the test definition to a file
F7 – Delete the selected queue
F8 – Clear the screen to start a new test

QUEUE EDITOR – Define Work Items, Parameters, and Worker Settings

The Queue Editor defines the work performed by a worker pool. A queue contains SQL statements, file references, or OS commands, along with optional parameter sets, pacing, and scheduling options. Workers draw tasks from queues during a test run.

Entry fields appear as `[_____]`.
Dropdown selections appear as `{ option }`.

Use arrow keys to move between fields and lines.

Queue Name

Enter the name of the queue. Use letters, digits, and underscores. Queues must be defined before referencing them in `WORKER`, `PACE`, `RUN`, or advanced scripting commands.

Work Items

Each line defines a single unit of work. A work item may be:

- a literal SQL statement
- a path to a `.sql` or command file
- an OS command, selected by changing the `Type` dropdown to `{ OS }`

For each work item, the editor shows: - the text or file reference
- the `Type` (SQL, OS, or FILE)
- an optional `AT` value

The `AT` field schedules execution at a specific offset from the test start (e.g., `30s`, `2m`). This is useful for injecting events such as DML operations, metadata updates, or intentionally expensive queries to simulate production spikes. For large-scale query replay, use the CLI command `QUEUE LIST` to auto-build work items from timestamped logs.

Press `Enter` to commit the current line.
Press `F7` to delete the highlighted line.
Press `F9` to browse for files to insert as work items.

Parameter Directory / Search Pattern

Optional. Enter a directory or wildcard pattern pointing to `.param` files. Each parameter row is substituted into placeholders `:1`, `:2`, etc., and becomes a separate execution of each work item.

Example parameter file: `east | 03`

Worker Connection Alias and Count

Choose the database alias for workers that run this queue. This field is a dropdown showing aliases defined by the `DB` command.

`Count` determines how many worker threads will use this queue.

Pace

Controls how fast the queue issues work. Examples:

- `0.5s` – wait 0.5 seconds between items
- `15%` – target percentage relative to other queues
- empty – no pacing applied

RowCount (N)

When enabled, TrueBench fetches all result rows from each query and counts them. This is helpful during validation to ensure queries return the expected volumes.

Note: RowCount introduces network, client, and driver overhead. For pure database performance testing, leave this disabled to avoid masking DB latency with client/network costs, especially on short queries.

Rows (Output Sample)

Specifies how many rows to display from each query result. Use this only when the test run type is SERIAL; parallel output from multiple workers can interleave unpredictably.

CLI mode supports per-worker output files, but within the TUI, this sample is mainly useful for initial correctness testing.

Keys

`F1` – Help
`F3` – Return to previous screen

`F7` – Delete current work item
`F9` – Browse for files
`Enter` – Add or commit field / work item

TUI Run Monitor

Overview
The TUI Run Monitor is the interactive display used by TrueBench to present real-time execution status while a test is running. It appears automatically when a test is launched from the CLI or from a TUI screen and remains active until the run completes or is terminated.

The run monitor is a read-only view with respect to test definitions. It does not modify queues, scripts, or parameters. Its purpose is to provide visibility into progress, activity, and runtime messages as the workload executes.

Screen layout
The screen is divided into three main regions arranged from top to bottom.

- The header displays high-level context for the run, including the script name, test name, run mode, elapsed time, remaining time (when applicable), and overall run status.
- The queue status section summarizes execution activity by queue. Each queue is listed with cumulative execution counts, error counts, zero-row counts, and the number of active workers currently running for that queue. Queue names are left-justified, and the list is indented for readability.
- The log output section displays a continuously updated stream of runtime messages. This includes echoed input, informational messages, warnings, errors, and system messages generated during execution.

A footer line displays function key assignments and navigation hints. Disabled keys appear visually distinct when not applicable.

Log output and color semantics
Log entries are displayed in chronological order and are automatically wrapped to the current screen width. Long messages are never truncated horizontally; instead, they are wrapped into multiple display lines.

Each log entry is assigned a semantic role such as informational output, input echo, warning, or error. Colors used to render these roles are controlled by settings in `truebench.profile` which may be viewed or modified with the **PROFILE** command. Adjusting profile color settings affects all TUI screens consistently, including the run monitor.

Log output continues to accumulate even when the screen is temporarily hidden or overlaid by another TUI view.

Navigation and scrolling
The log output area supports vertical navigation using the Up and Down arrow keys as well as Page Up and Page Down. Scrolling does not pause execution; it only changes the portion of the log currently visible.

A scroll position indicator shows which portion of the log buffer is currently displayed relative to the total number of retained log lines.

When the terminal window is resized, the run monitor automatically recomputes layout and rewraps log lines to fit the new dimensions.

Function keys
The run monitor provides a consistent set of function keys aligned with other TUI screens.

- F3 — Exit the run monitor after execution has completed.
- F4 — Request a soft stop, allowing in-flight work to finish gracefully.
- F5 — Request an immediate kill of all workers. Availability depends on DBMS and driver support; not all database platforms support hard termination.
- F6 — Emit an on-demand status snapshot into the log, summarizing current execution metrics.

Some function keys are disabled after execution completes or when a given action is not supported by the current environment.

Completion behavior
When execution finishes, the run monitor updates the overall status to indicate completion and preserves the final elapsed time. Log navigation remains available, allowing review of results before exiting the screen.

The run monitor exits only when explicitly requested by the user with the F3 key after completion.

Command Line Interface to Define Tests and Automate Testing

Basic Test Definition

These 4 commands are all you need to define to create a test. Tests begin with the **DEFINE** command to give your test a name and description and are initiated with the **RUN** command. Commands between those two commands define the work (via **QUEUE**) and the number of workers (via **WORKER**) statement. There are advanced test commands that can be added between **DEFINE** and **RUN** to further refine the test.

DEFINE — Start a new test giving a test name and description

Syntax:

`DEFINE <test_name> [description]`
Start a new test run and record its definition in the tracking database.

Description:

The DEFINE command initializes the runtime environment for a new test. It clears prior queues, variables, and counters, acquires a database lock, and creates a new entry in the **TestTracking** table. Subsequent commands such as QUEUE, PARAM, and WORKER will associate their activity with this test run.

Parameters:

- **test_name** — Unique identifier for the test. Appears in TestTracking.

- **description** — Optional short text describing the run.

Examples:

- `DEFINE nightly_run "Nightly benchmark of workload A"`
  Creates a new test named **nightly_run** with the provided description.

- `DEFINE smoke_test`
  Creates a new test named **smoke_test** with no description.

Notes:

- Only one test run is active at a time. Issuing DEFINE again will end the previous context and start a new one.

- The run ID assigned by the database is stored in the substitution variable `:runid` for later reference. THe other variables defined:
  `:testname`
  `:testdescription`

QUEUE – Define or extend a query queue

Syntax

```
QUEUE <qname> {AT <number>[s|m|h]} {SQL | SPLIT | LIST | OS | CONTROL}
[pattern | statement | stop...]
```

Description

The QUEUE command defines or extends a logical queue of work that will be executed during a test run by one or more worker threads.
Queues may contain SQL statements, operating system commands, or control instructions such as STOP and KILL. Queues are created automatically when referenced for the first time.

The QUEUE command may be issued multiple times for the same queue to append additional work. Queue execution behavior is controlled later by commands such as **WORKER**, **RUN**, **PARAM**, **ROWS**, and **ROWCOUNT**.

Four content modes are supported in addition to the default behavior:

- **SQL** – Default mode. The entire file is treated as a single SQL script block and executed exactly as written.
- **SPLIT** – The file is split at each semicolon (;) into multiple SQL statements. Each statement becomes a separate queue item named <filename>_1, <filename>_2, and so on.
- **LIST** – The file contains a list of SQL filenames, one per line. Each referenced file is read and added as a separate queue item.
- **OS** – Each non-comment line in the file represents an operating system command to execute on the client host.
- **CONTROL** - An explicit prefix to STOP or KILL statements to

Control instructions such as STOP, STOP.ME, KILL, and KILL.ME may be queued inline to influence test flow during execution. Not all DBMS or execution environments support hard kill semantics.

Parameters

- **qname**
  The name of the queue to create or extend. Queue names are case-insensitive.

- **AT <number>[s|m|h]**
  Optional delay before queued items become eligible for execution. Time units may be seconds (s), minutes (m), or hours (h).

- **REPORT | SQL | LIST | OS**
  Optional mode specifying how the target file or statement is interpreted.

- **pattern | statement | stop...**
  One of the following:
    - A file name or wildcard pattern (for example `queries/*.sql`)
    - An inline SQL or OS command
    - A control instruction such as `stop` or `kill`

Examples

`QUEUE Q1 AT 5s REPORT myscript.sql`
Queue the file `myscript.sql` as a single SQL block, starting 5 seconds into the test.

`QUEUE Q2 SPLIT queries/*.sql`
Split each `.sql` file into separate statements, each added as its own queue item.

`QUEUE Q3 LIST all_sql_files.txt`
Load SQL files listed in `all_sql_files.txt` into the queue. Each line in the file should contain a filename. Relative paths are resolved relative to the list file's directory.

`QUEUE Q4 OS os_commands.txt`
Add each line of `os_commands.txt` as a separate operating system command.

`QUEUE Q5 stop`
Queue a control instruction that stops all workers when this queue completes.

`QUEUE Q5 stop Q1`
Stop work in queue `Q1` when the work in queue `Q5` completes.

Notes

- A queue must be defined before it is referenced by **WORKER**, **PARAM**, **ROWS**, or **ROWCOUNT**.
- File paths in `QUEUE` commands are resolved relative to the issuing script's directory when used inside a script, or relative to the current working directory when issued interactively.
- Mixing SQL, OS, and control items in a single queue is allowed but discouraged for clarity.
- In SQL mode, statements are internally numbered using the `_n` suffix. Parameter matching via **PARAM** uses the root portion of the statement name.
- Queue contents and execution status are visible in the TUI run monitor and via commands such as [[SHOW]] or **LIST**, where available.

WORKER – Assign workers to queues

Syntax:

`WORKER <qname | *> <db_alias | OS> [count]`
Assigns one or more worker threads to execute queued commands.

Description:

The WORKER command defines which database connection or operating system context is used to execute queued statements. Multiple workers can be assigned to the same queue to enable concurrent execution.

Parameters:

- **qname** – Name of the queue, or * to apply to all queues.
- **db_alias | OS** – Database alias from the DB command, or `OS` for operating system commands.
- **count** – Optional number of workers (default = 1).

Examples:

- `WORKER Q1 TD_DB 2`
  Assigns two workers using database alias `TD_DB` to queue `Q1`.
- `WORKER Q2 OS`
  Assigns an operating-system worker to execute shell commands in queue `Q2`.

Notes:

- OS workers cannot execute SQL items.
- A separate worker thread is created for each queue, allowing independent execution and timing.

RUN – Execute queued workloads

Syntax:

`RUN {[<number>[s|m|h]]} {KILL} {SERIAL}`
Starts the workload execution using all defined workers.

Description:

The RUN command begins execution of all queues using their assigned workers. This utility supports three test execution models:

- **Fixed workload** – the default when the RUN command has no parameters. All workers start and execute the queries and commands in their queue and then stop. While this is the simplest execution model used by many other query drivers, concurrency is not maintained. Short queries will complete first, reducing overall concurrency to the remaining long-running queries.
- **Serial** – used to validate the basic performance of each query or command without workload competition. One queue is processed at a time with only one worker (the count specified on the WORKER command is ignored).

- **Fixed time period** – simulates a live, production workload with all defined workers processing all queues for a fixed duration. By adjusting the number of workers per queue and using the **PACE** command to control query arrival rate or workload percentage, you can create a realistic simulation of production behavior. When all queries (and their associated parameter sets from **PARAM**) have been executed, workers automatically loop to the first query and continue until the defined time period expires.

Parameters:

- **[s|m|h]** – Optional duration for timed runs (e.g., `60s`, `5m`, `1h`).

- **KILL** – When specified after a specified duration, in-flight queries are canceled when the time expires.

- **SERIAL** – Executes each queue one at a time with one worker.

Examples:

- `RUN`
  Executes all queues once in parallel.

- `RUN 15m`
  Executes for 15 minutes (timed run).
- `RUN .25h`
  Executes for 15 minutes, same as 15m or 900s

- `RUN 5m KILL`
  Executes for 5 minutes and cancels in-progress queries at expiry.

- `RUN SERIAL`
  Executes queues one at a time, single-threaded.

Notes:

- All queues must have at least one worker.

- If any worker fails to connect within 30 seconds, execution continues with available workers.

- The **KILL** option is intended to stop long-running queries at the end of a timed run.
  Its implementation (and effectiveness) is DBMS-driver-specific.
  If unsupported by your DBMS, manual cancellation may be required.

EXIT / QUIT – Terminate TrueBench

Syntax:

`EXIT [rc]` `QUIT [rc]`

Terminate TrueBench and perform a clean shutdown.

Description:

The **EXIT** command terminates TrueBench and performs an orderly shutdown of all active resources, including the metadata tracking database.

If profile settings were modified during the session, TrueBench may prompt to save those changes before exiting, depending on whether the session is interactive.

Parameters:

- **rc** – Optional. Integer return code to use when terminating TrueBench.
  If omitted, the return code defaults to `0`.

Behavior:

When EXIT or QUIT is executed, TrueBench performs the following steps in order:

- Closes the metadata tracking database if it is open
- Checks whether profile settings were modified
- If running interactively and the profile is dirty:
  - Prompts whether to save the changes
  - Saves the profile only if explicitly confirmed
- Terminates TrueBench with the specified return code

Interactive vs Batch Mode:

- In **interactive mode**, TrueBench prompts to save modified profile settings.
- In **batch or script mode**, no prompts are issued and profile changes are discarded.
  To persist profile changes in batch mode, the script must explicitly issue: `PROFILE SAVE`

Examples:

- `EXIT`
  Exit TrueBench with return code 0.

- `EXIT 2`
  Exit TrueBench with return code 2.

- `QUIT`
  Exit TrueBench (synonym for EXIT).

- `EXIT ?`
  Display short help for the EXIT command.

Notes:

- `EXIT` and `QUIT` are synonymous.
- `EXIT` overrides any remaining commands in a script.

- When running inside automation or shell pipelines, the returned exit code may be used to detect success or various failure conditions.

Immediate Commands

These commands perform their action immediately, even if they are between the **DEFINE** and **RUN**, but are most often used during interactive development of tests, or before/after tests.

ECHO – Print or write text with variable substitution

Syntax:

```
ECHO [> file | >> file] text
ECHO <<EOF | delim=label
ECHO {ON | OFF}
```

Description:

The ECHO command writes text to the console or a file, or turns echo of input on or off. It supports variable substitution (e.g., `:runid`, `:testname`, `:os`) and multi-line text entry using heredoc syntax.

Parameters:

- **text** – The text to print or write. Variables beginning with `:` are replaced by their current values.
- **> file** – Write to the specified file, overwriting any existing content.
- **>> file** – Append text to the specified file.
- **<<EOF** or **delim=label** – Enter multi-line text interactively until the given delimiter label is entered.

Examples:

- `ECHO "Run :testname completed with :errors errors"`
  Displays a summary message on the console.
- `ECHO > results/summary.txt "Test :testname finished at :os"`
  Writes a line to *results/summary.txt*.
- `ECHO <<END`
  (Then type multiple lines, ending with "END" on a line by itself.)
- `ECHO delim=STOP`
  (Equivalent heredoc syntax using custom delimiter STOP.)

Notes:

- Directories in file paths are created automatically if they do not exist.

- Variable substitution supports test and environment variables such as `:runid`, `:testname`, `:cd`, `:homedir`, and `:os`.

- Empty ECHO statements output a blank line.

CD – Change or Reset Current Working Directory

Syntax:

`CD {path}` Change or reset the current working directory for QueryDriver.

Description:

The **CD** command changes the active working directory used by QueryDriver to locate SQL scripts, parameter files, and output files.

If no path is provided, the directory is reset to the QueryDriver home directory (the parent directory of the `query_driver` installation folder).

Parameters:

- **path** – Optional. Relative or absolute path of the directory to switch to.
  If omitted, QueryDriver resets to its home directory.

Examples:

- `CD /tmp/queries`
  Change to the `/tmp/queries` directory.

- `CD testcases`
  Change to the `testcases` subdirectory under the current directory.

- `CD`
  Reset to QueryDriver's home directory.

- `CD ?` In addition to the normal short help, the current directory is displayed

Notes:

- Relative paths are resolved from the current working directory.
- QueryDriver maintains its own internal "current directory" state to ensure consistency across commands like `QUEUE`, `PARAM`, and `ROWS`.

TRACE - Control terminal output

Syntax:

`TRACE {OFF | ON | VERBOSE }`
Controls how much information is displayed on the console while always logging all messages to the database.

Parameters:

- **OFF** - Display only critical messages and summary output (level 0).
- **ON** - Default setting. Display basic command feedback (level 1).
- **VERBOSE** - Display detailed internal activity such as worker start/stop (level 2).

Examples:

- `TRACE OFF`
  Display only error messages.
- `TRACE VERBOSE`
  Display detailed TrueBench processing messages. This was only set up for development and debugging of TrueBench.
- `TRACE`
  Display the current trace setting.

SQL – Execute an ad-hoc SQL statement or script

Syntax

```
SQL <db_alias | me> <pattern | statement> {;} {PARAM <values...>} {> | >> filename}
```

Runs a SQL command or executes one or more script files against a database connection.

Description

The **SQL** command executes either:

- a literal SQL statement, or
- one or more `.sql` files matched by a wildcard pattern.

The SQL is executed against the database alias previously defined using **db**, or against the internal metadata database using `me`. If the query is successful, the :errorcode variable is set to 0. Otherwise, it will attempt to return the errorcode from the DBMS or -1.

Optional parameters (`PARAM <values>`) allow substituting `:1`, `:2`, ... inside inline SQL or inside script files before execution.

Parameters

- **db_alias**
  A previously defined database alias created using **db**, or `me` for the internal `testtracking.db`.

- **pattern | statement**
  A filename pattern (e.g., `scripts/*.sql`) **or** a literal SQL statement. If the pattern expands to one or more files, each file is executed in order.

- **;**
  Ends the SQL portion of the command when redirection is used. (See Notes below.)

- **PARAM <values...>**
  Optional list of parameter values.
  Values substitute `:1`, `:2`, etc. within the SQL or script text before execution.

- **> filename**
  Redirects output to a new file.

- **>> filename**
  Appends output to an existing file.

Examples

Run a simple statement:
```
SQL me SELECT runid, test_name FROM TestTracking
```

Count rows:
```
SQL mytd SELECT COUNT(*) FROM customer
```

Execute all `.sql` scripts in a directory:
```
SQL mytd myqueries/*.sql
```

Execute a script and redirect output:
```
SQL mytd myqueries/*.sql ; > results/all_queries.txt
```

Inline SQL with parameters:
```
SQL td SELECT * FROM customer WHERE region=':1' PARAM east
```

Multiple parameters:
```
SQL td SELECT * FROM sales WHERE region=':1' AND month=:2 PARAM east 03
```

Execute a script with parameter substitution:
```
SQL td ddl/create_customer.sql PARAM DEMO_DB
```

Notes

- Using `me` connects to the internal metadata database in **read-only** mode.
- Output is **pipe-delimited**, with embedded newlines removed for easy import into Excel or log parsers.
- **When using redirection (> or >>), the SQL must be terminated with a semicolon.**
  This prevents ambiguity between:
  - `>` used as part of a SQL expression (e.g., `WHERE x > 5`)

- `>` used as a redirection operator
  Example:

  ```
  SQL td SELECT * FROM t WHERE x > 5 ; > output.txt
  ```

- When not using redirection, the SQL **does not need** to end with `;`.
- Parameter substitution occurs **before** execution and applies to inline SQL as well as to the contents of matched `.sql` files.
- Each result set is preceded by a short header identifying the source statement or file executed.

BEFORE_SQL – SQL executed before every SQL command

Syntax:

```
BEFORE_SQL [alias | * ] { <SQL-STATEMENT> | LIST | CLEAR }
```

Description:

Registers one or more SQL statements that will be executed **before every** `SQL <alias> <statement>` command.
The statements run in the **same session and cursor** as the main SQL, allowing session initialization such as query bands, TEMP tables, or SET SESSION options.

The `SQL-STATEMENT` applies to the immediate **SQL** command.
To define commands to execute by workers before each SQL statement, use the **BEFORE_QUERY** statement.

Parameters:

- **alias** - The database alias established by the **DB** command
- ** * ** - Applies to all database aliases except me

Notes:

- If you have defined multiple aliases for multiple logons on the same DBMS, you can use the * command to apply to all of them.
- If you want a **BEFORE_SQL** setting for the **ME** alias, you must explicity specify it.

Examples:

Set a session query band before every SQL command for the database alias `td`:
`before_sql td SET QUERY_BAND='truebench_prep' FOR SESSION;`

List entries across all database aliases:
`before_sql * list`

Remove the aliases defined for `td`:
`before_sql td clear`

OS – Execute an operating system command

Syntax:

`OS <command string> {> | >> <filespec>}`
Executes an operating system command with optional output redirection.

Description:

The OS command executes a command through the host operating system shell. The standard output and standard error streams can be redirected to files using > (overwrite) or >> (append).

Variables such as `:runid`, `:testname`, or `:os` are substituted before execution, enabling test-specific output or automation steps. If the execution is successful, the :errorcode variable will be set to 0. Otherwise it will be set to the return code from the OS.

Parameters:

- – The OS command and arguments to run.
- **> filename** – Redirect output to file (overwrites existing).
- **>> filename** – Append output to file.

Examples:

- `OS ls -l scripts/tpch/queries > dirlist.txt`
  For **Linux**, writes directory listing to *dirlist.txt*.
- `os dir scripts\tpch\queries > dirlist.txt`
  For **Windows**, writes the directory listing to *dirlist.txt*
- `OS cp :homedir/transaction_save/* :homedir/transactions`
  For **Linux**, copy in a fresh set of transaction files for the next test.
- `OS rem :homedir\temp\*`
  For **Windows**, clear out the temp directory for the next test.

Notes:

- TrueBench commands are not case sensitive; **Linux commands and file names are case sensitive**.
- **OS commands are executed verbatim by the host operating system.**
- **Windows uses \ as a path separator; Linux uses /.**
  For OS commands, use the correct separator for your platform.

- TrueBench does **not** automatically rewrite path separators for OS commands.
- Output redirection supports absolute or relative paths.
- Variable substitution occurs before execution.
- On Linux, full-screen TUI programs such as `vi` or `less` **will not work**.
- Environment variable changes made with `set` (Windows) or `export` (Linux) are **not preserved** across OS command executions.
- `OS cd` commands do **not** change the current directory used by TrueBench; use the TrueBench `CD` command instead.
- For complex Windows command chains, you may explicitly use `cmd /c` if needed.

LIST — Display test metadata, SQL statements, aliases, or available drivers

Syntax:

```
LIST {SQL | DB | DRIVERS | ODBC} {runid | runid1 runid2 | -number} [WHERE <condition>]
```

Description:

Displays information from the TrueBench metadata database (`testtracking.db`) or related runtime state.
When no keyword is provided, `LIST` reports summary information from the **TestTracking** table.

Parameters:

- **SQL** – Displays logged SQL or command `INPUT` lines for a single test run (from **TestTrackingLog**).
  Only one runid may be specified.
  Example: `LIST SQL 21`
- **DB** – Lists all currently defined database connection aliases.
  Example: `LIST DB`
- **DRIVERS** – Lists all supported database drivers and their capabilities.
  Example: `LIST DRIVERS`
- **ODBC** - Lists all registered ODBC drivers in the client OS. Example: `LIST ODBC`
- **runid** – Specific test run to display.
- **runid1 runid2** – Inclusive range of test run IDs.

- **-number** – Displays the most recent *n* runs (e.g., `LIST -4`).

- **WHERE** – Filters rows from the **TestTracking** table (e.g., `WHERE test_name LIKE '%join%'`).

Examples:

- `LIST -5`
  Lists the five most recent test runs.

- `LIST -30 WHERE notes LIKE '%final%'`
  Lists test in last 30 tests where notes contain “final”.

- `LIST 100 110`
  Lists test runs 100 through 110.

- `LIST SQL 120`
  Lists logged SQL `INPUT` lines for run 120.

- `LIST DB`
  Lists all currently defined database aliases.

- `LIST DRIVERS`
  Lists all supported database drivers with timeout/cancel support indicators.

Notes:

- The **SQL** option supports only a single runid and ignores WHERE conditions.

- When no subcommand is provided, `LIST` defaults to listing tests from the **TestTracking** table.

NOTE – Add or update notes for a test run

Syntax:

```
NOTE [runid] {add} {description}
```

Description:

Allows you to record or append a note to the `TestTracking` table for the specified `runid`. Useful for documenting test conditions, environment details, or observations made after a run has completed.

If **add** is specified, the text will be appended (comma separated) to the current note entry for that run.

If no description text is provided, the existing notes for the run will be cleared.

Examples:

- `NOTE 120 Describe test environment and changes`

- `NOTE 120 add Included new stats on customer table`
- `NOTE 120`
  (clears existing notes for runid 120)

Notes:

- Variable substitution is supported in the note text.
- Notes are stored in the `notes` column of the `TestTracking` table.
- Changes are committed immediately.

Advanced Test Commands

These commands are slightly more advanced and modify the test behavior.

LIMIT – Set maximum query runtime per queue

Syntax:

`LIMIT <qname | *> <time>[s|m|h]` Sets a maximum allowed runtime for queries executed by a queue.

Description:

The LIMIT command specifies how long a query in a given queue may run before it is automatically canceled (if supported by the underlying DBMS driver).
If the driver does not support active cancellation, the query continues until completion, but its elapsed time will still be recorded in the results.

Parameters:

- **<qname | *>** – The name of the queue to apply the limit to, or `*` to apply to all queues.

- **[s|m|h]** – The maximum time allowed per query, expressed in seconds, minutes, or hours (e.g., `60s`, `5m`, `1h`).

Examples:

- `LIMIT * 60s`
  Set a 60-second limit for all queues.

- `LIMIT Q2 2m`
  Restrict queries in queue *Q2* to 2 minutes maximum.

Notes:

- The effectiveness of the LIMIT command depends on the driver implementation. For example:
  - **SQLite** supports per-query cancellation via `set_progress_handler()`.
  - **Teradata** and some ODBC drivers do not support active cancellation, but elapsed times are still logged.

- When a query exceeds its limit, the worker thread attempts to cancel it, marks the query as failed, and continues with the next workload item.

PARAM – Associate parameter (.param) files with queued queries

Syntax:

`PARAM <qname> [pattern] {<delimiter>|<tab>}`
Load .param files for queries in a queue.

Description:

Associates .param files with queued queries in the specified queue.
Each file should contain rows of parameters for repeated query execution.
The first row is always treated as data (no headers).
Columns are referenced as positional variables like `:1`, `:2`, etc.

Parameters:

- **qname** – Name of the queue containing SQL or OS commands.
- **pattern** – Optional string to match query names (e.g., `sales_` loads all queries beginning with `sales_`).
- **delimiter** – Optional field separator (default `,`); use `<tab>` for tab-delimited files.

Examples:

`PARAM Q1 scripts/*.param`
All parameter files in the scripts directory ending with the suffix of .param will be used for Q1 using the default comma as a delimiter. Note that if the scripts directory had been specified by the queue command as `scripts\*.sql`, then query01.sql would use query01.param, query02.sql would use query02.param, etc. If the SQL file doesn't have a matching .param file, then no parameter substitution will occur.
`PARAM Q1 parameters/*.param <tab>` Parameter files will be taken from the parameters directory and the columns will be delimited with the tab character
`PARAM Q2 scripts/sales_*.param |`
Associates all `.param` files matching `sales_*.param` in the scripts directory with queue `Q2`. The columns are pipe delimited.

If the scripts/query01.sql had the statement:

```
select count(*) from invoices where inv_date=':1' and district = :2;
```

and the param specification referenced a directory containing query01.param with the following content:

```
2025-01-15,10
2025-01-20,10
2025-01-20,15
```

Then the following executions would occur:

```
select count(*) from invoices where inv_date='2025-01-15' and district = 10;
select count(*) from invoices where inv_date='2025-01-20' and district = 10;
select count(*) from invoices where inv_date='2025-01-20' and district = 15;
```

Notes:

- Paths are relative to the current working directory.
- Each `.param` file must have the same number of columns per row.
- Parameter values are substituted using positional placeholders like `:1`, `:2`.
- Use `TRACE VERBOSE` to verify parameter binding during test execution.

PACE – Control query release rate or workload share per queue

Syntax:

`PACE <qname | *> [n<S|M|H> | n%] { qname ... n% ... }`
Defines pacing rules for one or more queues, either by time interval or workload percentage.

Description:

The PACE command regulates how frequently queries are released from each queue or how much of the total workload a queue contributes. It supports two pacing models:

- **Interval pacing** – limits how often queries are released (e.g., every 2 seconds).
- **Percentage pacing** – limits the number of queries in a queue to a percentage of the total executed

This command is typically used in **timed** test runs to maintain steady concurrency and a realistic mix of workload activity.

Parameters:

- **<qname | * >** – Queue name to apply pacing, or * to apply to all queues.
- **n<S|M|H>** – Release interval in seconds (`s`), minutes (`m`), or hours (`h`).
- **n%** – Percentage of total workload to assign to the queue.

Examples:

- `PACE * 2s`
  All queues release queries every 2 seconds.
- `PACE Q1 2s Q2 20% Q3 40%`
  Queue Q1 releases every 2 seconds; Q2 and Q3 execute 20% + 40% = 60% of the queries. If there is a Q4, while the other queues are constrained, it will be free to execute as many queries as it can.

Notes:

- The **total percentage across all queues** must not exceed 100%.

- If you specify a percentage for all defined queues, you may not saturate the DBMS platform because a long running query in a queue could cause all of the other queues to wait to maintain the percentage. If the goal is to saturate the platform, don't put percentages on all queues.

- When you specify timed release and there are insufficient workers to initiate a new query execution at the specified time, a "deficit will be reported at the end of the test for that queue indicating you need to increase the workers.

- When you specify percentages on queues, at the end of the test, the total seconds of delay will be reported as the workers wait for workers in other queues to meet their percentage. Large wait times on a queue indicates the level of concurrency was reduced and you may need more workers on the other queues.

- Pacing affects the releasing of queries on a queue and is most useful for preventing queues with short tactical queues from executing an unrealistic number of queries during a test. It can also simulate anticipated events during the workload tests like call center queries, killer queries, or the arrival of data batches.

- The count of workers on a queue sets the maximum number that can be executing concurrently and is an alternative method of modeling the production workload. Both can be used together to model the workload.

- If you have two queues with similar running queries and set the pace of one to 50%, they are likely to execute the same number of queries since the pace constrained queue is 50% of the total executed in both queues. For Q2 to execute half the queries for Q1 + Q2, set the pace on Q2 to 33%.

ROWS – Configure query result retrieval and output

Syntax:

`ROWS <qname | *> [max_rows] [> filename | >> filename]` Set how many rows are returned by each query and where they are written.

Description:

The ROWS command controls how many rows are returned by queries executed from worker threads.
For performance testing, set `max_rows` to 0 (the default) to avoid client and I/O overhead that can mask true DBMS performance.

Output files may include substitution variables in the **file name** portion such as `:queue`, `:workerno`, `:runid`, or `:testname`.
Example: `results_:queue_:workerno.txt` produces a unique file per queue and worker.

Important:
- Do not use query-specific variables (e.g. `:queryid`, `:queryname`) since files are opened once per worker, not per query.
- Variables cannot appear in the directory path, which must exist before the test begins.

Parameters:

- **qname** – Name of a specific queue, or * to apply to all queues
- **max_rows** – Maximum rows to fetch per query, default 10 (0 = no fetch)
- **filename** – Optional file to write results (overwrite or append)

Examples:

```
ROWS Q1 10
```
Return up to 10 rows per query from queue `Q1` to the terminal.

```
ROWS * 25 > results/:queue_:workerno.txt
```
Each worker writes 25 rows per query to its own file, e.g. `results/Q1_1.txt`, `results/Q2_2.txt`.

```
ROWS * >> logs/all_rows.log
```
Append results from all queues to the shared log file.

Notes:

- When using output redirection (> or >>), the directory must already exist.
- Filenames may include `:queue` and `:workerno` substitutions (e.g. `results/:queue_:workerno.txt`).
- Do **not** use query-specific variables in filenames, since the file is opened once when the worker starts.
- Output is **pipe (|) delimited** with embedded newlines removed for Excel compatibility. You can copy/paste results directly into Excel and use "Text to Columns → Delimited → |".

ROWCOUNT – Enable post-execution row counting for validation

Syntax:

```
ROWCOUNT <qname | *>
```
Enable row counting for one or all queues after each query executes.

Description:

Causes workers to count the number of rows returned by each query and record that value in the `TestResults` table for validation or comparison purposes.

Parameters:

- `qname` – Name of the queue to enable row counting on.

- * – Enables row counting for all queues.

Examples:

ROWCOUNT *
Enables row counting for all queues.

ROWCOUNT Q1
Only queue `Q1` will record row counts after each query.

Notes:

- Row counting is primarily used to confirm query correctness, not for performance measurement.
- Without **ROWCOUNT** setting, queries are executed but no rows are returned to the client platform where TrueBench is running. If you are measuring DBMS performance, don't burden your measurements with the latency and bandwidth associated with returning rows to the client running TrueBench to count. The performance of your client platform can also impact the measurement of DBMS performance.
- If queries run more than 5 - 10 seconds, then the impact of **ROWCOUNT** is small. On the other hand, if your test includes tactical queries running less than 1 second, the latency and bandwidth could easily mask the underlying performance of the DBMS.

BEFORE_WORKER - Statements to execute by worker before test starts

Syntax

```
BEFORE_WORKER <qname | *> <SQL-statement>
```

Description

Associates SQL statement(s) with one or more queues to execute at the start of the test. Each worker assigned to that queue will execute the statement(s) **once**, immediately before beginning its main workload loop.

This is typically used to: - Set the default database or search path
- Initialize session-level variables
- Apply Optimizer or session settings
- Log session metadata

The SQL is executed only once per worker, **not once per query**. The command to define SQL to execute before each query is: **BEFORE_QUERY**.

Parameters

- **qname** - Name of the queue receiving the BEFORE_WORKER rule.
- * - Apply the rule to **all queues** currently defined in this test. If additional queues are defined later, they will not be assigned this query.
- **SQL-statement** - Any single SQL statement. Multi-statement blocks are not allowed.

Behavior

- BEFORE_WORKER rules are **ephemeral**.
  They are automatically cleared when a new **DEFINE** starts.
- If a SQL error occurs during BEFORE_WORKER execution, the worker stops and registers an error in the test logs.
- The statement executes on the worker's database connection, using the same alias defined by **WORKER**.
- Multiple BEFORE_WORKER statements for the same queue are executed in the order they were defined.

Examples

Set the default database for all workers:
```
BEFORE_WORKER * DATABASE finance_db
```

Enable a Teradata session mode for queue 'q1' only:
```
BEFORE_WORKER q1 SET SESSION CHARACTER SET UNICODE
```

Record a worker-level startup timestamp:
```
BEFORE_WORKER q2 INSERT INTO audit_log VALUES (CURRENT_TIMESTAMP, 'worker start')
```

BEFORE_QUERY – Execute SQL before each query in a queue

Syntax

```
BEFORE_QUERY <qname | *> <sql-statement>
```

Description

Defines a SQL statement that will be executed **before each query** issued by workers in the specified queue.

This is typically used for setting **query bands**, **session attributes**, or other per-query metadata that must be refreshed before each statement.

The SQL runs using the **same database alias** that the worker uses for the queue.

Parameters

- `qname` — A queue name defined with **queue**.
- * — Apply the SQL to **all queues** defined so far (except OS queues).
- `sql-statement` — The SQL text to execute.

Behavior

- The SQL defined by BEFORE_QUERY is executed **immediately before each query** in the queue.
- If the SQL fails, the worker logs the error and **skips that query**.
- BEFORE_QUERY definitions are **ephemeral** and are cleared automatically when a new **define** begins a test.

Variables available for substitution

These are expanded before each execution:

- `:queue` — queue name
- `:workerno` — worker number
- `:queryid` — internal query identifier
- `:queryname` — query file name
- All global variables defined by **set**

Example: `BEFORE_QUERY q1 SET QUERY_BAND = 'queue=:queue worker=:workerno query=:queryname' FOR SESSION`

Apply to all queues: `BEFORE_QUERY * SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED`

AFTER_QUERY – Execute SQL after each query in a queue

Syntax

```
AFTER_QUERY <qname | *> <sql-statement>
```

Description

Defines a SQL statement that will be executed **after each query** issued by workers in the specified queue.

Use this for cleanup actions, auditing, temporary object maintenance, or any post-processing logic that depends on the results of the query that was just executed.

The SQL runs using the **same database alias** that the worker uses for the queue.

Parameters

- `qname` — A queue name defined using **queue**.
- * — Apply the SQL to **all queues** (except OS queues).
- `sql-statement` — The SQL text to execute.

Behavior

- The AFTER_QUERY SQL runs **immediately after each query** executes.
- If the AFTER_QUERY SQL fails, the worker logs the error and continues.
- AFTER_QUERY definitions are **ephemeral** and are cleared automatically when **define** starts a new test.

Variables available during substitution

These variables are substituted into the AFTER_QUERY SQL:

Worker / queue context - `:queue` — queue name
- `:workerno` — worker number
- `:queryid` — query identifier
- `:queryname` — query file name
- All global variables set using **set**

Post-query result variables (new) - `:returncode` — the return code from the query
- `:errormsg` — the message text, if any
- `:responsemsec` — elapsed time in milliseconds
- `:rows` — number of rows returned (if any)
- `:timestamp` — timestamp at the moment AFTER_QUERY executes

Example: audit insert

```
AFTER_QUERY q1 INSERT INTO audit(msg, rc, msec)
VALUES('done :queryname', :returncode, :responsemsec)
```

Example applied to all queues

```
AFTER_QUERY * INSERT INTO eventlog(worker, q, rc, rows)
VALUES(:workerno, :queue, :returncode, :rows)
```

Analyzing Test Results

Using the built-in client level tracking

TdBench uses an internal SQLite database to record - one line for every test in the **TestTracking** table - one line for every input and log message in the **TestTrackingLog** table - one line for every query execution in every test in **TestResults** table

Some of that data is retrieved via the **LIST** command. More directly, you can use the **SQL** or **SELECT** commands to pass SQL to SQLite to perform test analytics.

METADATA – TrueBench Metadata Tables and Structure

TrueBench records all test activity in a local SQLite database stored in: metadata/truebench.db

This metadata captures session locking, test summaries, per-query results, and detailed log messages. See also the SQLite reference in `HELP SQLITE` for examples of how to query or maintain this database.

TestTracking Table

The `TestTracking` table contains one row for each executed test (`RUN`).
This provides the high-level summary shown in CLI and TUI reports.

Columns: - `RunId` – Unique run identifier - `TestName` – Name assigned to the test - `Description` – Optional description from the TUI editor - `StartTime` – UTC timestamp when the test began - `StopTime` – UTC timestamp when the test completed - `QueriesExecuted` – Total queries executed by all workers - `ErrorCount` – Number of queries with non-zero return codes - `ZeroRowCount` – Queries that returned no rows - `WorkerCount` – Number of workers used for this run - `Notes` – Optional free-form notes

Example: - `RunId=1013 TestName='LatencyCheck' QueriesExecuted=3200 ErrorCount=2`

TestTrackingLog Table

The `TestTrackingLog` table stores detailed chronological messages generated during a test. This includes informational messages, warnings, and errors.

Columns: - `RunId` – Links back to `TestTracking` - `LogLineNo` – Sequential log line number - `LogTime` – Timestamp of the message - `LogType` – `INFO`, `WARN`, `ERROR`, etc. - `LogText` – Full text of the log entry

Example: - `RunId=1013 LogLineNo=42 LogType='WARN' LogText='Zero rows returned'`

TestResults Table

The `TestResults` table contains one row for every query executed by every worker during a test. This is the primary source for latency and throughput analysis.

Columns: - `RunId` – Parent test run identifier - `QueryName` – Name of the query or SQL file - `QueueName` – The queue the query belonged to - `ReturnCode` – 0 for success, non-zero for errors - `StartTimestamp` – When the query execution began - `StopTimestamp` – When the query completed - `ResponseMsec` – Milliseconds between start and stop - `Rows` – Number of rows returned - `Parms` – Parameter values after variable substitution

Example: - `QueryName='order_by_id' ResponseMsec=481 Rows=1`

SessionLock Table

The `SessionLock` table prevents multiple TrueBench processes from modifying the metadata at the same time. Only a single row is stored.

Columns: - `id` – Primary key - `active` – 1 when the lock is held, 0 when not - `user` – Username holding the lock - `host` – Hostname of the locking machine - `timestamp` – UTC timestamp when the lock was acquired

Example: - One active lock row indicates TrueBench is running: `active=1 user='doug' host='WIN11' timestamp='2025-11-29T13:40:22Z'`

Table Relationships

The metadata tables relate through the `RunId` column:

- `TestTracking` is the parent record for each test run
- `TestTrackingLog` contains many log entries per run
- `TestResults` contains many query results per run
- `SessionLock` is independent and controls metadata access

Example workflow: - During command processing input lines are - A row is added to `TestTracking` when a run begins
- During execution, workers write rows to `TestResults`
- Log messages are recorded in `TestTrackingLog`
- At test completion, the `TestTracking` row is updated with final counts

Inspecting Metadata

Useful examples (run via `sqlite3`):

- `SELECT RunId, TestName, StartTime FROM TestTracking ORDER BY RunId DESC` Shows recent test runs.

- `SELECT QueryName, ResponseMsec FROM TestResults ORDER BY ResponseMsec DESC LIMIT 10`
  Displays the slowest queries.

- `SELECT LogTime, LogText FROM TestTrackingLog WHERE RunId=1013`
  Shows all log messages for a specific run.

Resetting Metadata

TrueBench recreates the database automatically if it is removed.

- `rm metadata/truebench.db`
  Deletes all stored test history and logs.

Use caution when removing the file, as these results cannot be recovered.

SQLITE – SQLite Commands for Viewing and Maintaining Metadata

SQLite is a lightweight, self-contained SQL database engine that stores all data in a single file and requires no server, which makes it ideal for TrueBench to record test history, results, and execution logs with zero deployment overhead. Originally created by D. Richard Hipp in 2000, SQLite has become one of the most widely used databases in the world, embedded in applications such as Chrome, Firefox, Android, iOS, and countless desktop tools. Its strengths are simplicity and portability, but its limitations include lack of true client/server concurrency, no user accounts or permissions system, and performance constraints when handling very large datasets or highly parallel write workloads.

For the TrueBench SQLite driver, see [[sqlite3]].

Syntax:

SQLite queries can be executed with the TrueBench `SQL me ...` command or the alias `SELECT`, with your favorite **SQL GUI**, or by downloading the `sqlite3` command-line tool.

The metadata database is located at:
`metadata/truebench.db`

Description:

TrueBench stores all test history, logs, and results inside a SQLite database.

This topic provides a quick reference for common SQLite statements used to inspect or manage metadata, including `SELECT`, `JOIN`, `GROUP BY`, `CREATE VIEW`, and `DELETE`.

Full SQLite reference manual:
https://www.sqlite.org/lang.html

Information on the TrueBench usage of SQLite can be viewed by issuing `HELP METADATA`.

SELECT – Basic Query Syntax:

General form of a SQL `SELECT`:
`SELECT columns FROM table WHERE condition GROUP BY columns HAVING condition ORDER BY columns`

Examples: - `SELECT RunId, TestName, StartTime FROM TestTracking ORDER BY RunId DESC LIMIT 5`
Lists the most recent test runs in descending order.

- `SELECT RunId, AVG(ResponseMsec) AS AvgMs FROM TestResults GROUP BY RunId ORDER BY AvgMs`
  Shows average response time per test run.

- `SELECT t.TestName, r.QueryName, r.ResponseMsec FROM TestTracking t JOIN TestResults r ON t.RunId = r.RunId WHERE r.ReturnCode <> 0 ORDER BY r.ResponseMsec DESC`
  Displays queries that returned non-zero return codes.

- `SELECT RunId, COUNT(*) AS Cnt FROM TestResults GROUP BY RunId HAVING Cnt > 500`
  Shows tests that executed more than 500 queries.

CREATE VIEW – Save Reusable Queries:

Syntax:
`CREATE VIEW viewname AS SELECT ...`

Examples: - `CREATE VIEW SlowQueries AS SELECT RunId, QueryName, ResponseMsec FROM TestResults WHERE ResponseMsec > 1000`
Stores a view of queries slower than 1 second.

- `SELECT * FROM SlowQueries LIMIT 10`
  Reads from the view.

- `DROP VIEW SlowQueries`
  Removes the view.

DELETE – Removing Rows (e.g., runaway test cleanup):

Always test deletes with a `SELECT` first.

Examples: - `SELECT * FROM TestTracking WHERE RunId = 31`
Checks rows before removal.

- `DELETE FROM TestResults WHERE RunId = 31`
  Removes all per-query results for the test.

Transaction-safe pattern: - `BEGIN TRANSACTION`
Starts a manual transaction. - `DELETE FROM TestResults WHERE RunId = 31`

Performs cleanup. - `ROLLBACK`
Undo if results look incorrect. - `COMMIT`
Make cleanup permanent.

Listing Tables and Views:

Examples: - `SELECT name FROM sqlite_master WHERE type='table' ORDER BY name`
Lists all tables.

- `SELECT name FROM sqlite_master WHERE type='view' ORDER BY name`
  Lists all views.

- `SELECT type, name FROM sqlite_master WHERE type IN ('table','view') ORDER BY name`
  Lists both.

Listing Columns of a Table or View:

Use `PRAGMA table_info`:

- `PRAGMA table_info(TestTracking)`
  Shows column names, positions, and types for the `TestTracking` table.

Additional Useful Metadata Queries:

Examples: - `SELECT QueryName, RunId, ResponseMsec FROM TestResults ORDER BY ResponseMsec DESC LIMIT 10`
Shows the slowest queries across all test runs.

- `SELECT TestName, ZeroRowCount FROM TestTracking ORDER BY ZeroRowCount DESC`
  Ranks tests with the most zero-row results.

- ```
  SELECT t.TestName, l.LogTime, l.LogText FROM TestTrackingLog l
  JOIN TestTracking t ON t.RunId = l.RunId
  WHERE l.LogType='ERROR' ORDER BY l.LogTime
  ```
  Displays all error-type log messages.

- `SELECT QueueName, AVG(ResponseMsec) FROM TestResults GROUP BY QueueName ORDER BY AVG(ResponseMsec)`
  Shows average response time by queue name.

SELECT – Execute SQL using the default alias

Syntax:

```
SELECT <sql_text>
```
```

Description:

Executes the given SQL text using the default database alias (`me`).
`SELECT` is a convenience alias for:

`SQL ME <sql_text>`

This allows quick, ad-hoc SQL execution without specifying an explicit alias.

Example:

- `SELECT * FROM TestTracking ORDER BY RunId DESC LIMIT 5`
- `SELECT RunId, TestName FROM TestTracking WHERE ErrorCount > 0`
- `SELECT logtype, count(*) from TestTrackingLog group by logtype`
- `SELECT QueryName,  ResponseMsec from TestResults where RunId = 21 order by QueryName`

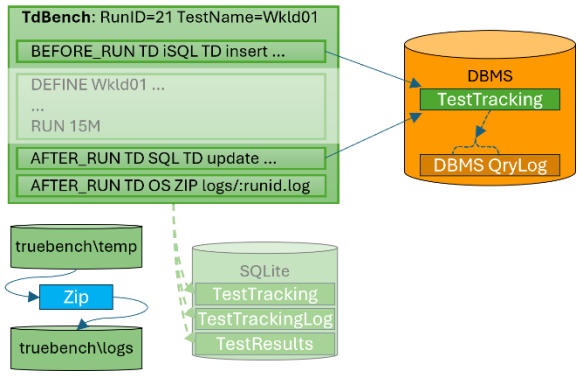
Notes:

- Supports full variable substitution (`:1`, `${VAR}`, `:runid`, etc.).
- May be used inside script files (`.tdb`) just like any other command.

Linkage to Host DBMS Query Logging

TdBench records the performance of every query execution in its metadata. For comparing the performance of two DBMSs or platforms, that may be sufficient.

For use in development, DBMS engineering, or quality control, it isn't sufficient to know that a query took "n" seconds to execute. It is useful to know what the DBMS was doing to execute the query. The query plan, DBMS CPU usage or I/O performance requires usage of the query logging and resource usage tracking inside DBMS system tables.



With dozens or hundreds of tests performed over several months, it can be challenging to compare a run just completed with a particular test execution a week ago.

Commands are provided to execute before and after a test to maintain a test tracking table on the host DBMS. It will also duplicate notes applied to the internal metadata database to the host DBMS's test tracking table for easy reference of settings or observations of the test execution.

BEFORE_RUN – Execute a command immediately before workers start

Syntax:

`BEFORE_RUN {<alias> <truebench-command> | LIST | CLEAR}`
Registers a TrueBench command that will run *after RUN is issued but before workers begin execution*.

Description:

The BEFORE_RUN command allows you to insert one or more TrueBench commands that execute **after the RUN command is parsed**, but **before any worker thread starts running**. This is typically used to initialize state in an external database or operating system environment prior to workload execution.

Commands registered with BEFORE_RUN execute *in the same variable-substitution environment* as normal runtime commands.

Parameters:

- **alias** - Identifies the alias of the workers on the test that this BEFORE_RUN applies to

- **truebench-command** — Any valid TrueBench command except flow-control commands such as `GOTO`, `IF`, and nested `RUN`.

The command is stored verbatim and executed later.

- **LIST** - List out current definitions for BEFORE_RUN.

- **CLEAR** - Wipe out current definitions for BEFORE_RUN.

Examples:

- `BEFORE_RUN td sql td insert into truebench.testtracking(runid, testname, testdescription) values(:runid, ':testname', ':testdescription');`
  Inserts a row into an external test-tracking table before workers start.

- `BEFORE_RUN td os rm -f /tmp/testlogs/*`
  Cleans a directory before the test begins.

Notes:

- When a test is defined, the first alias used by a worker defines the alias for the test. The alias on the BEFORE_RUN associates statements to be executed when tests are executed against that platform. Most often, any alias from the BEGIN_RUN will match the alias in a specified SQL statement.

- Commands are stored without substitution; variable expansion occurs at execution time.

- BEFORE_RUN commands run *after* the test definition is complete and just before worker startup.

- Multiple BEFORE_RUN commands are executed in the order they were defined.

AFTER_RUN – Execute a command after RUN completes and summary is displayed

Syntax:

`AFTER_RUN {<alias> <truebench-command> | LIST | CLEAR}`
Registers a TrueBench command that will run *after workers finish and after final summary output*.

Description:

The AFTER_RUN command allows you to record test completion details, update external metadata tables, or perform cleanup actions after all workload processing is finished.

AFTER_RUN commands execute at the very end of `RUN`, during the `finalize_run()` phase—after worker threads have closed and after the summary and LIST -4 are produced.

Parameters:

- **alias** - Identifies the alias of the workers on the test that this AFTER_RUN applies to

- **truebench-command** — Any valid TrueBench command except control-flow commands (`GOTO`, nested `RUN`, `IF`). The command is stored literally and executed later using normal parsing and substitution.

- **LIST** - List out current definitions for AFTER_RUN.

- **CLEAR** - Wipe out current definitions for AFTER_RUN.

Examples:

- `AFTER_RUN sql td update truebench.testtracking set stoptime=current_timestamp, queries=:queries, errors=:errors, workers=:workers where runid = :runid;`
  Updates the testtracking row on the host DBMS (td) with final statistics.

- `AFTER_RUN os zip logs/:runid.log.zip :homedir/temp/*`
  Compresses generated logs after the test completes for each test.

Notes:

- When a test is defined, the first alias used by a worker defines the alias for the test. The alias on the AFTER_RUN associates statements to be executed when tests are executed against that platform. Most often, any alias from the AFTER_RUN will match the alias in a specified SQL statement.

- AFTER_RUN commands execute *after* workers stop and test results are finalized.

- Variable substitution (e.g., `:runid`, `:queries`) uses final test values.

- Multiple AFTER_RUN commands run sequentially in the order defined.

AFTER_NOTE — Execute commands immediately after a NOTE is applied

Syntax:

`AFTER_NOTE { <alias> <statement> | CLEAR | LIST }`
Registers commands to execute immediately after a `NOTE` command. The system variable `:note` contains the text of the last NOTE.

Description:

`AFTER_NOTE` allows you to define additional commands that run whenever a NOTE is added to the test record. This is typically used to synchronize NOTE text into an external TestTracking table or log.

The variable `:note` is automatically set to the text of the most recent NOTE command and is available for variable substitution inside AFTER_NOTE statements.

Parameters:

- **alias** - Identifies the alias for the runid that was specified on the note statement
- **statement** — Any TrueBench command (SQL, OS, ECHO, etc.) to execute after NOTE.
- **CLEAR** — Removes all stored AFTER_NOTE commands.
- **LIST** — Displays all currently stored AFTER_NOTE commands.

Examples:

```
AFTER_NOTE td sql td update truebench.testtracking set note=':note' where runid=:runid;
```
Propagates the NOTE text into the TestTracking table on the external database.

```
AFTER_NOTE CLEAR
```
Removes all AFTER_NOTE commands.

```
AFTER_NOTE LIST
```
Shows currently stored AFTER_NOTE commands.

Note:

- The intent of the after_note is to keep documentation in sync between testtracking and the host DBMS When TrueBench is supporting tests to multiple DBMS platforms, the note posted to the local testtracking table should be reflected in the testtracking table on the host DBMS where the test was executed. That means that the alias for the AFTER_NOTE command will be the same as the alias in the SQL xxx Update statement.

Scripting Commands

These commands allow you to generalize the definition of tests and then reuse code for different situations.

Basically, you could save a defined test as a TdBench command file, either from the TUI editor or coding CLI commands directly into a file. You can then execute that test with a single **EXEC** statement.

A more advanced usage would be to use some variables in the defined test such as the number of workers for a queue or the duration of the test. You could then provide those parameters on the **EXEC** command line.

Even more advanced would create a new command file that executes other command files for a test suite. That new "master" command file could execute the same test multiple times, each time changing the numbers of workers to increase the concurrency for each execution.

Finally, you can use the other scripting commands to prompt the user, set variables, and change the execution with if and goto statement. ### Programming Constructs

Scripting in TrueBench allows your test definitions to branch, loop, and interact with users or environment variables.
These constructs give you control-flow features similar to simple shell or batch scripting.

The core statements are: `SET`, `READ`, `IF`, `DO/END`, and `GOTO`.
Each section below provides syntax, behavior, and usage notes.

SET – Assign a Variable

The `SET` command assigns a value to a variable. Variables created via `SET` are available to subsequent statements and SQL files.

Syntax:
```
SET name = value
```

Examples:
```
SET region = US_East
SET threshold = 200
```

Variables created by SET:
- Do not override built-in variables such as `:runid` or `:testname`.
- Can be referenced later using `${name}` or as `:name` inside SQL (depending on substitution context).

Use **`SET`** for more details.

READ – Prompt the User or Pull Default Values

The `READ` command interactively asks the user for a value if one is not already present in the environment or in previously SET variables.

Syntax:
```
READ name : prompt string
```

If the variable already exists (via environment or SET), it is not reprompted unless `READ` was declared with the `REQUIRED` option (future extension).

Example:
```
READ dbpass : Enter database password:
```

Use **`READ`** for more details.

IF – Conditional Execution

The `IF` command tests a variable or expression and conditionally executes a block.
An IF may be followed by optional `ELSE` and must terminate with either a single command or a `DO`/`END` block.

Syntax:
```
IF expression THEN command
or
IF expression THEN DO
... commands ...
END
```

Expressions currently supported:
- Variable equality: `${var} == value`
- Variable inequality: `${var} != value`
- String match: `${var} = value` (exact match)
- Existence: `${var}` (true if non-empty)

Example:
```
IF ${region} == US_East THEN DO
  SET db = east1
END
```

Use **`IF`** for more details.

DO / END – Multi-line Command Blocks

`DO`/`END` encloses a multi-line sequence of commands.
This allows grouping several statements under IF, or for future looping constructs.

Syntax:
```
THEN DO
```

```
  commands...
END
```

or standalone:

```
DO
  commands...
END
```

A DO/END block may contain `SET`, `READ`, `IF`, or `GOTO`.

Use **`DO`** or `[[END]]` for more details.

GOTO – Unconditional Jump

`GOTO` jumps forward to either a label or to a subsequent test name.
Labels begin with a colon and appear at the start of a line.

Syntax:

```
GOTO labelname
GOTO testname
```

Label examples:

```
:RetryBlock
:BranchForUS
```

Important notes:
- GOTO only jumps **forward**, never backward.
- GOTO cannot enter another test.
- GOTO inside DO/END exits the block.

Use **`GOTO`** for more details.

Summary

Programming constructs improve flexibility in TrueBench test definitions by letting you:
- Define variables (`SET`)
- Prompt for user-defined values (`READ`)
- Execute conditionally (`IF`)
- Group commands (`DO`/`END`)
- Jump ahead (`GOTO`)

These commands make tests more dynamic and reusable across environments, DBMS targets, and user workflows.

EXEC – Execute a test script with optional parameters

Syntax:

```
EXEC <filename> [arg1 [arg2 ...]]
```

Runs another test script, optionally substituting argument variables.

Description:

The **EXEC** command executes another QueryDriver script as a complete test, optionally passing arguments that can control variables such as the number of workers, query pacing, or run durations.

Arguments supplied after the filename are substituted as positional variables `:i1`, `:i2`, `:i3`, etc., within the executed file.
Each level of EXEC has its own local variable scope, so values from one test do not affect another.

INCLUDE is an alias for this command but is typically used to insert reusable definitions within a larger test, while **EXEC** is oriented toward executing full workloads or sequences of tests.

Parameters:

- – Path to the test script (.tdb or .inc) to execute.
- **[arg1 [arg2 ...]]** – Optional arguments mapped to `:i1`, `:i2`, etc. inside the executed file.

Examples:

- `EXEC my_workload.tdb 2 1 3`
  Executes *my_workload.tdb* with three queues using 2, 1, and 3 workers respectively.
- `EXEC my_workload.tdb 4 2 6`
  Re-executes the same workload with higher concurrency (12 workers total).
- `EXEC ../tests/smoke.tdb`
  Runs a predefined smoke test located in another directory.

Notes:

- Each EXEC runs independently, creating a new runid when a `DEFINE` is executed.
- Arguments (`:i1`, `:i2`, ...) are scoped to the current EXEC level and do not persist.
- Nested EXEC commands are supported up to `MAX_INCLUDE_DEPTH` levels.
- A brief INFO log (level 2) notes when each EXEC begins and ends.
- For inserting shared setup blocks during test definition, use **INCLUDE** instead.
- path separators (/ or) will be adjusted based on what is valid for your OS

INCLUDE – Reuse command sequences when defining tests

Syntax:

`INCLUDE <filename> [arg1 [arg2 ...]]`
Inserts and executes commands from another file to simplify test setup.

Description:

The **INCLUDE** command allows you to reuse common sets of statements (for example, hundreds of `QUEUE` definitions) across multiple test scripts.
It acts as a simple macro facility that copies and executes the contents of the specified file during test definition.

Relative paths inside the include file are resolved with respect to that file's location. Both the OS and session working directories are restored when the include completes.

Each invocation of INCLUDE has its own argument scope. Optional arguments passed on the command line are available as positional variables `:i1`, `:i2`, etc., within the included file.

The **EXEC** command is an alias that uses the same mechanism but is intended for test execution rather than setup.

Parameters:

- – Path to the command file to include.
- **[arg1 [arg2 ...]]** – Optional arguments mapped to `:i1`, `:i2`, `:i3`, etc.

Examples:

- `INCLUDE common/setup.inc`
  Reuses standard initialization commands during test definition.
- `INCLUDE queueset.inc A 15m`
  Expands `:i1` and `:i2` within *queueset.inc* to customize queue naming or duration.

Notes:

- INCLUDE is useful for reusing test definitions, not for running complete workloads. Use **EXEC** for test execution.
- Nested INCLUDE statements are allowed up to `MAX_INCLUDE_DEPTH` (default = 10).
- Included commands are not logged individually, but a brief INFO message (level 2) marks the start and end of each include.
- Directory context is automatically restored after completion.

- path separators (/ or \ ) will be adjusted based on what is valid for your OS

SET – Define, clear, or list variables

Syntax:

```
SET var [=] [value]
```

Description:

Defines or clears a user-defined variable.
If called without arguments, lists all current **system** and **user** variables. System variables are read-only and maintained automatically by QueryDriver; user variables are created or modified with `SET` or `READ`.

Parameters:

- **var** – Variable name (Do not preceed with a colon).
- **value** – Value to assign. If omitted, the variable is cleared.
- **=** – Optional separator for readability (`SET :var = value`).

Examples:

- `SET userid dbadmin`
- `SET region = us-east`
- `SET userid` *(clears variable)*
- `SET` *(lists all variables)*

Notes:

- System variables such as `:os`, `:cd`, `:runid`, and `:testname` cannot be modified.
- Use `HELP variables` for information on standard variables and substitution capabilities.

READ – Prompt for variable input

Syntax:

```
READ :var [prompt text]
```

Description:

Prompts interactively for a value and stores the result in a variable. If no prompt text is supplied, QueryDriver displays a generic prompt based on the variable name.

Parameters:

- **:var** – Variable name (colon optional).

- **prompt text** – Optional text displayed before reading input. The text may include other variables that will be expanded before display.

Examples:

- `READ userid Please enter your login ID:`
  Assigns user entry to the variable userid which can be substituted in other commands as :userid.

- `READ password Enter database password for :userid:`
  Assuming the user entered a logon id (e.g. johnsmith), that value would be substituted in the prompt:
  … Enter database password for johnsmith:
  If they just pressed Enter on the prior prompt for :userid without entering a password, the prompt would be:
  … Enter database password for :userid

You could address the failure to enter a userid with:
`if :userid not set then goto missing_userid`

Notes:

- System variables listed in **SYSTEM_VARIABLES** (e.g., `:os`, `:runid`) cannot be changed.

- **NOTE** Don't use `read :password Enter your password` in your include file since if :password is alredy set, the current value will become the variable name.
- Variable substitution applies only to the prompt text.

IF – Conditional execution with THEN/ELSE

Syntax:

```
IF <left> <operator> <right> THEN <statement> [ELSE <statement>]
IF :var { IS SET | NOT SET } THEN <statement> [ELSE <statement>]
IF <filename> {EXISTS | MISSING} THEN <statement> [ELSE <statement>]
```

Description:

Evaluates a condition and, if true, executes the `THEN` statement; otherwise the optional `ELSE` statement. Variables and `${ENV}` are expanded before evaluation.

Parameters:

- **left / right** – Operands for comparison (after substitution).
- **operator** – One of: =, !=, >, >=, <, <=.
- **:var IS SET** – True if var exists and is non-empty in user or system scope.

- **:var NOT SET** – True if var is missing or empty.
- **EXISTS** - True if the file is
- **statement** – Any single command (`ECHO`, `READ`, `GOTO`, `DEFINE`, etc.).

Example:

- `IF :password NOT SET THEN READ password Enter your password:`
  ... some more code
  `ELSE ECHO Using existing password`

- `IF ${password} NOT SET THEN READ logonpw else if :password IS SET then logonpw = :password ELSE echo ERROR - No available password`

Notes:

- If both operands parse as numbers, numeric comparison is used; otherwise string comparison (case-sensitive).
- `ELSE` may appear inline on the same line, or as a separate command immediately after `IF`.
- Use `\` at the end of a line to split an inline `THEN ... ELSE ...` across multiple lines.

ELSE – Alternate execution if prior IF was false

Syntax:

```
ELSE <statement>
```

Description:

Executes the statement only if the immediately preceding `IF` evaluated to false.

Example:

- `IF :x > 10 THEN ECHO Large`
  ... some more code
  `ELSE ECHO Small`

Notes:

- Must directly follow an `IF` (no intervening commands).
- Resets the IF/ELSE state after execution; chaining multiple `ELSE` lines is not supported.
- The `<statement>` may be any single command and follows normal substitution rules.

GOTO – Skip forward to a label or test

Syntax:

```
GOTO <labelname | testname>
```

Description:

Skips forward within the current INCLUDE or EXEC script until a matching `LABEL` or `DEFINE` statement is found.

Example:

- `... some code`
  `GOTO finalize` *(later in same script)*
  ... some more code
  `LABEL finalize`

Notes:

- Only forward jumps are allowed; backward GOTO is ignored.
- GOTO works only within the same INCLUDE file—it does not cross into nested INCLUDE or EXEC files.
- Useful for conditional branching with the `IF` command.

LABEL – Define a target for GOTO

Syntax:

```
LABEL <name>
```

Description:

Marks a point in the current INCLUDE or EXEC script that can be referenced by a `GOTO` command. Labels are local to the script file in which they appear and cannot be referenced across INCLUDE boundaries.

Example:

- `... some code`
  `GOTO finalize` *(later in same script)*
  ... some more code
  `LABEL finalize`

Notes:

- Labels are case-insensitive.
- Backward GOTO jumps are not supported.
- Duplicate labels in the same file are ignored with a warning.

DO – Begin a multi-line block (terminated by END)

Syntax:

```
THEN DO
ELSE DO
END
```

Description:

Marks the beginning and end of a multi-line statement block used within an `IF` or `ELSE` control structure. All commands between `DO` and `END` are executed sequentially when the associated condition is true (for `THEN DO`) or false (for `ELSE DO`).

Example:

- ```
  IF :ready = yes THEN DO
  ECHO Starting test
  RUN
  END
  ... some more code
  ELSE DO
  ECHO Skipping test
  END
  ```

Notes:

- `DO` and `END` must appear alone on their lines.
- Nested `DO ... END` blocks are not supported.
- The block executes in the same variable and directory context as the surrounding script.
- You may still use single-line `THEN` or `ELSE` forms for brief logic.

PROFILE – View or update persistent environment settings

Syntax:

```
PROFILE { [key] [value] }
```

Show or update persistent profile settings and display the location of the active `truebench.profile` file.

Description:

The `PROFILE` command manages TrueBench's persistent configuration, stored in the file `truebench.profile` located in the user's TrueBench home directory.

Profile settings are: - loaded automatically at startup - applied immediately after any update - used by both the CLI and the full-screen TUI

Profile entries are stored in the form: `<key>=<value>`

Supported Settings

SQL tagging settings
These settings control whether TrueBench injects identifying tags into each SQL statement. Tags make it easier to find benchmark statements in host DBMS query logs and relate them to test results.

Values: `yes` or `no`
```

- `insert_query_name` – when `yes`, injects `tdb=<queryname>` into SQL (recommended)
- `insert_runid` – when `yes`, injects `runid=<runid>` into SQL (default `no`)

Notes: - `insert_query_name=yes` helps identify each benchmark query in DBMS query history. - `insert_runid=yes` is important for DBMSs that aggregate query metrics over time windows (for example Azure Query Store). It allows separating runs before and after tuning changes, even when multiple tests occur within the same summary interval.

CLI colors
These affect command-line (non-TUI) output.

Values: any supported terminal color name, optionally prefixed with `bright_`
Examples: `white`, `red`, `bright_cyan`

- `color.cli.input` – color used for echoed input commands
- `color.cli.output` – color used for normal command output
- `color.cli.error` – color used for error messages
- `color.cli.code` – color used for code examples and command references
- `color.cli.link` – color used for links to other help files

TUI colors
These affect all full-screen TUI screens (run monitor, file picker, queue editor, help viewer). TUI colors are defined as: `foreground:background`

Values: `fg:bg`, where each is a supported terminal color name; `bright_` prefix allowed
Examples: `bright_white:black`, `black:white`, `bright_cyan:black`

- `color.tui.text` – default body text
- `color.tui.info` – informational messages (INFO)
- `color.tui.input` – input fields or echoed command lines
- `color.tui.warning` – warning messages
- `color.tui.error` – error messages
- `color.tui.header` – screen headers and titles

- `color.tui.footer` – footer lines and status bars
- `color.tui.label` – labels preceding values (for example `Script:` or `Status:`)
- `color.tui.value` – values associated with labels
- `color.tui.separator` – separator lines between screen regions
- `color.tui.keys` – enabled function-key labels
- `color.tui.keys.disabled` – disabled function-key labels
- `color.tui.selected` – selected row or cursor highlight
- `color.tui.link` – help links
- `color.tui.code` – inline code fragments in help text

Directories
These settings control where TrueBench reads scripts and writes output.

Values: a directory path
Examples: `C:\truebench\scripts`, `/home/user/truebench/logs`

- `dir.scripts` – directory searched for `.tdb` INCLUDE scripts
- `dir.logs` – directory where logs and redirected output files are written
- `dir.temp` – temporary directory for generated scripts and intermediate files
- `dir.metadata` – directory containing the TrueBench tracking database
- `dir.flag` – directory monitored for stop/pause/kill control flags

Operational settings
These settings control refresh frequency and log retention limits.

Values: integer numbers

- `watchdog.status_interval` – seconds between periodic status updates during long runs
- `tui.log_line_limit` – maximum number of log lines retained in TUI displays

Examples

Display current profile `PROFILE`

Update a directory location `PROFILE dir.scripts C:\truebench\scripts`

Enable SQL tagging for host query log analysis `PROFILE insert_query_name yes`
`PROFILE insert_runid yes`

Color Templates

There are two scripts in the setup directory that may help with setting up the colors for light and dark screens. They will set the colors and then save your profile.

`exec setup\theme_dark.tdb`
or: `exec setup\theme_light.tdb`

Notes

- Profile changes take effect immediately for new CLI output and new TUI screens.
- Only the `bright_` prefix is supported; no `dim_` colors are recognized.
- TUI colors are semantic; changing a role affects all screens consistently.
- Unrecognized keys are rejected with a validation error and are not persisted.
- Profile files are rewritten safely using a temporary file and atomic replace.

Drivers to connect to your DBMS

Database Drivers

Overview

A TrueBench database connection requires:

- a DBMS-specific driver
- a URL / hostname (or other endpoint identifier)
- logon credentials

TrueBench supports three categories of database drivers:

1) Known Python database drivers (installed into the TrueBench `.venv`)
2) Generic Python DB-API drivers (installed into the TrueBench `.venv`)
3) ODBC database drivers (installed in the operating system)

Python Drivers (Preferred)

For many DBMS platforms, TrueBench uses a Python driver.

If the required Python module is not installed, TrueBench can offer to install it into the TrueBench `.venv` at the time you define your database alias.

Python drivers are preferred because: - installation can be automated by TrueBench (for known drivers) - they avoid OS-level binary installs (ODBC drivers) - they are more portable across Windows/Linux/macOS/WSL - they often provide better performance for tactical queries than ODBC

Generic Python DB-API Drivers

If your DBMS platform is not supported as a known built-in driver, you may still be able to connect using a generic DB-API 2.0 Python module.

This uses the `pydbapi` driver and requires:

- the DBMS-specific Python module installed into `.venv` (manual install via pip)
- the module name specified in the DB alias definition

Example:

```
db pg pydbapi url=localhost user=myuser password=? module=psycopg2
```

Notes: - The DBMS module must expose `connect()` and behave as a DB-API compatible driver. - TrueBench does not auto-install modules for `pydbapi`. Install the module manually into the TrueBench virtual environment. - The `PREPARE` command is not supported for the generic Python driver. TrueBench performs variable substitution before execution, so normal `sql` and benchmark runs are not impacted.

ODBC Drivers

For many databases (and some PC data sources like Excel, Access, and CSV/text files), TrueBench can connect using ODBC.

ODBC connectivity in TrueBench has two components:

1) `pyodbc` (Python interface inside the TrueBench `.venv`)
2) an ODBC driver installed in your operating system

TrueBench can offer to install `pyodbc` automatically if it is missing.
However, the OS-level ODBC driver must be installed manually.

ODBC drivers are installed outside TrueBench and registered in the OS ODBC subsystem.

For help installing and validating ODBC drivers, see: - [[odbc]] - INSTALL.md

The `odbc` driver name is passed as a parameter to `pyodbc` along with the URL, logon credentials, and optionally other driver-specific parameters.

DB Aliases (How Workers Connect)

Worker threads executing queries during a benchmark run, and immediate **sql** commands used to prepare test environments, connect using a DB alias.

A DB alias defines: - the driver - the URL / endpoint - logon credentials - optional connection parameters

Aliases are established with the **db** command.

Examples:

Known Python driver: `db td teradata url=myhost user=myuser password=?`

Generic Python driver: `db xyz pydbapi url=myhost user=myuser password=? module=some_dbapi_module`

ODBC driver: `db srv odbc url=myhost user=myuser password=? driver="ODBC Driver 17 for SQL Server"`

Once the alias exists, workers and SQL commands use that alias name:

```
sql td select current_date
```

Why Multiple Aliases Matter

You can define multiple aliases for the same DBMS platform or server. This is useful to:

- isolate test query execution from credentials of a higher-privileged setup user
- separate credentials for different workloads, enabling DBMS workload management policies

- separate credentials to monitor different workloads in real-time using DBMS workload tools

Important: Always Validate Aliases

After defining an alias, always run:

```
validate <alias>
```

This confirms: - the alias works - credentials are correct - basic SQL execution succeeds

It also measures baseline latency from this TrueBench client to the DBMS, which helps you understand how much of your measured response time is non-DBMS overhead (network jitter, round-trip latency, etc).

Viewing Installed Drivers

To see which drivers are installed and available in your TrueBench environment:

```
list drivers
```

This shows: - Installed Drivers (available now in this environment) - Other Supported Drivers (supported but not installed yet)

Driver-Specific Help and Setup Integration

To get help for a specific driver, use:

```
help <driver>
man <driver>
```

To get driver information (including installed status), use:

```
list driver <driver>
```

Some DBMS platforms also provide a setup directory for optional host-side TestTracking and query log correlation. This assists reporting and diagnosis by allowing you to join DBMS query history to TrueBench `runid` values.

Look in:

```
setup/<dbms>/README.md
```

Supported Databases (Alphabetical)

The following drivers are supported by this TrueBench release:

- **azure** — Microsoft Azure SQL Database / SQL Server connectivity
- [[bigquery]] — Google BigQuery
- **databricks** — Databricks SQL
- **greenplum** — VMware Greenplum (PostgreSQL protocol)
- [[ibm_db2]] — IBM Db2 / Db2 Warehouse (includes IIAS / appliance deployments)

- [[ibm_netezza]] — IBM Netezza Performance Server
- [[odbc]] — Generic ODBC (Excel, Access, CSV/text sources, and DBMS drivers via ODBC)
- **oracle** — Oracle Database (includes Exadata deployments)
- **pydbapi** — Generic Python DB-API 2.0 driver (module-driven, manual pip install)
- [[redshift]] — Amazon Redshift
- **snowflake** — Snowflake
- [[sqlite3]] — SQLite (local file-based database used for TrueBench metadata)
- **teradata** — Teradata Vantage and VantageLake (Python driver)
- [[teradata_odbc]] — Teradata Vantage and VantageLake (ODBC driver)

Release Disclaimer and Feedback

Not all database connections and driver configurations have been tested in this TrueBench release.

Database connectivity is often environment-specific due to: - firewalls and VPNs - SSL and certificate policies - SSO requirements - client network placement - DBMS platform versions and security settings

If you use TrueBench with a database platform, please provide feedback via GitHub and contribute improvements where possible, including: - corrections to connection parameters or installation instructions - enhancements to driver documentation - improvements to `setup/<dbms>` scripts for host DBMS query log reporting

Azure SQL (Python Driver)

Overview

TrueBench driver name (used in the DB command): `azure`

This driver connects to Microsoft Azure SQL platforms (commonly Azure SQL Database and Azure SQL Managed Instance). It uses a Python SQL Server client library and typically does not require an OS-level ODBC driver.

If you specifically need ODBC behavior or must match other ODBC tools, use [[odbc]] with an appropriate Microsoft ODBC Driver for SQL Server (if installed).

Installation Behavior

When you use `azure` in the `DB` command to define an alias used to connect to Azure SQL, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- wrong server name or port
- wrong database name
- wrong user / password (or Azure AD authentication requirements)
- firewall rules blocking access to the server
- corporate VPN / proxy restrictions
- encryption or certificate trust settings required by policy

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- SQL Server Management Studio (SSMS)
- Azure Data Studio
- sqlcmd
- DBeaver / DataGrip / VS Code SQL tools

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example (recommended alias: `az`):

```
db az azure server=myserver.database.windows.net database=mydb user=myuser password=?
```

Notes: - `server=` is typically a fully qualified Azure SQL hostname - `database=` is required - `password=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

```
validate az
```

This validates: - server hostname correctness and connectivity - authentication and login - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and Azure SQL - fastest achievable query response time via this client running TrueBench

This baseline helps separate performance impacts that are NOT DBMS execution time (especially for sub-second tactical queries), such as network latency and jitter.

It also provides a fairness check: when comparing two DBMS platforms, ensure the network overheads are equivalent across platforms — otherwise benchmark results may be misleading.

Important Benchmark Notes (Azure SQL)

Azure SQL performance can be influenced by environment features unrelated to SQL text:

- serverless tiers may auto-pause and introduce cold-start latency
- resource governance and throttling can affect throughput during sustained loads
- plan caching and buffer caching can cause large differences between first-run and steady-state execution

Strong recommendations: - run a warm-up phase before recording benchmark timing - run multiple iterations - report distribution statistics (p50 / p95 / max), not only averages - do not compare single-run timing numbers

Common Connection Parameters

You may append additional connection parameters as `name=value` pairs. These are passed through to the Azure SQL Python connector.

Common examples: - `server=<hostname>` - `port=1433` - `database=<dbname>` - `user=<user>` - `password=<pwd>` (or `password=?`) - `encrypt=true` - `trustservercertificate=false`

Some environments use Azure AD authentication or alternate methods. If your organization requires Azure AD auth, consult internal guidance for the correct connector parameters and approved authentication flow.

Setup Directory (Benchmark Reporting)

Many benchmarking teams maintain a host-side TestTracking table inside the benchmarked DBMS to support automated reporting and correlation by RunID (not just timestamps).

TrueBench can optionally maintain a host-side TestTracking table inside Azure SQL to support reporting workflows that join benchmark RunID to Azure-side telemetry.

If you want to integrate TrueBench RunID with Azure-side reporting, see:

`setup/azure/README.md`

This directory contains scripts to create: - a host-side TestTracking table (optional) - reporting queries/views joining Azure telemetry to runid

Troubleshooting

Cannot import Azure SQL connector modules The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md.

Authentication errors Verify: - user/password - database name is correct - Azure firewall rules allow your client IP or network - whether your org requires Azure AD authentication

Encryption / certificate errors Some environments require encryption and certificate validation. Verify: - `encrypt=` and `trustservercertificate=` settings - certificate trust policy and required root CA configuration

Network failures Verify the Azure SQL server is reachable from this client environment. If a familiar SQL tool cannot connect from the same machine, TrueBench will not be able to connect.

See Also

- [[drivers]]
- **db**
- **validate**
- [[odbc]]
- INSTALL.md

Google BigQuery (Python Driver)

Overview

TrueBench driver name (used in the DB command): `bigquery`

This is the preferred BigQuery driver for TrueBench. It uses the Google Cloud BigQuery Python client and does not require any OS-level ODBC driver.

BigQuery connectivity is different from most DBMS platforms because authentication is typically based on Google Cloud credentials (service accounts, application default credentials, SSO), not a simple user/password.

Installation Behavior

When you use `bigquery` in the `DB` command to define an alias used to connect to BigQuery, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- missing or incorrect Google Cloud credentials
- wrong project ID or dataset reference
- insufficient IAM permissions
- BigQuery API not enabled for the project
- corporate proxy restrictions blocking outbound access

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- Google Cloud Console (BigQuery UI)
- `bq` CLI
- DBeaver / DataGrip / VS Code SQL tools (BigQuery adapters)

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example (recommended alias: `bq`):

```
db bq bigquery project=my-gcp-project dataset=my_dataset location=US
```

Notes: - `project=` is the Google Cloud project ID - `dataset=` is commonly used as the default dataset for unqualified table references - `location=` should match where your dataset(s) reside (for example US or EU)

Authentication Notes (Critical for BigQuery)

BigQuery typically uses one of these authentication mechanisms:

- Application Default Credentials (ADC), often via `gcloud auth application-default login`
- Service account JSON key file
- Workload identity / platform identity (common on GCP compute)

If your organization uses service accounts, a common pattern is to set:

- `GOOGLE_APPLICATION_CREDENTIALS=<path to JSON key file>`

Credential handling is environment-specific. If you do not already have a working BigQuery client workflow, consult your organization's cloud admin guidance before benchmarking.

Validate Connectivity (Do this next)

After defining an alias, always run:

`validate bq`

This validates: - credentials and project access - ability to submit and run a basic query - basic end-to-end query execution through TrueBench

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and the BigQuery service - fastest achievable query response time via this client running TrueBench

This baseline helps separate performance impacts that are NOT DBMS execution time (especially for sub-second tactical queries), such as network latency and service overhead.

It also provides a fairness check: when comparing two DBMS platforms, ensure the network and client overheads are comparable — otherwise benchmark results may be misleading.

Important Benchmark Note: BigQuery Query Cache

BigQuery supports query result caching. If an identical query is re-run and the cached result remains valid, BigQuery may return cached results rather than executing the query.

For benchmarking, you may want to disable query caching so each query truly executes.

If query cache remains enabled, repeated benchmark runs can become dominated by cache reuse rather than actual DBMS execution capability.

Benchmarking guidance: - confirm whether query caching is enabled in your environment - disable cached results when strict execution testing is required - avoid repeated identical queries unless cache behavior is intentionally being measured

Common Connection Parameters

You may append additional connection parameters as `name=value` pairs. These are passed through to the BigQuery client configuration.

Common examples: - `project=<gcp project id>` - `dataset=<default dataset>` - `location=<US|EU|...>`

If your environment supports multiple authentication modes, you may also use parameters related to credentials (service account JSON path, token behavior, etc.), depending on how your BigQuery connector is implemented.

Because authentication is environment-specific, consult your cloud admin guidance for approved methods.

Setup Directory (Benchmark Reporting)

Many benchmarking teams maintain a host-side TestTracking table inside the benchmarked DBMS to support automated reporting and correlation by RunID (not just timestamps).

TrueBench can optionally maintain a host-side TestTracking table in BigQuery to support reporting workflows that join benchmark RunID to BigQuery-side telemetry.

If you want to integrate TrueBench RunID with BigQuery-side reporting, see:

`setup/bigquery/README.md`

This directory contains scripts to create: - a host-side TestTracking table (optional) - reporting queries/views joining BigQuery telemetry to runid

Troubleshooting

Cannot import BigQuery client modules The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md.

Authentication errors Verify: - credentials are available in this environment - IAM permissions allow query execution - project ID is correct - BigQuery API is enabled in the project

Unexpectedly fast results Likely caused by query result caching. Confirm whether query caching is enabled and disable cached results for benchmark runs where strict execution testing is required.

See Also

- [[drivers]]
- **db**
- **validate**
- INSTALL.md

Databricks (Python Driver)

Overview

TrueBench driver name (used in the DB command): `databricks`

This driver connects to Databricks SQL using the Databricks SQL Connector for Python. Databricks authentication typically uses: - server hostname - HTTP path - access token

If you specifically need ODBC behavior or must match other ODBC tools, use [[odbc]] with a Databricks ODBC driver (if installed).

Installation Behavior

When you use `databricks` in the `DB` command to define an alias used to connect to Databricks, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- incorrect server hostname (workspace URL)
- wrong HTTP path to SQL warehouse
- invalid access token or token expiration
- workspace network restrictions

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- Databricks SQL Web UI
- DBeaver / DataGrip / VS Code SQL tools

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example (recommended alias: `dbx`):

```
db dbx databricks server_hostname=myworkspace.cloud.databricks.com
http_path=/sql/1.0/warehouses/<id> access_token=?
```

Notes: - `server_hostname=` is the Databricks workspace hostname - `http_path=` identifies the SQL warehouse endpoint - `access_token=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

```
validate dbx
```

This validates: - connectivity and authentication to Databricks - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and Databricks SQL - fastest achievable query response time via this client running TrueBench

Common Connection Parameters

Common examples: - `server_hostname=<hostname>` - `http_path=<path>` - `access_token=<token>`

Refer to Databricks SQL connector documentation for additional parameters supported by your environment.

Setup Directory (Benchmark Reporting)

If you want to integrate TrueBench RunID with Databricks-side reporting, see:

`setup/databricks/README.md`

This directory may contain scripts and notes for integrating benchmark run metadata with Databricks query history.

Troubleshooting

Cannot import Databricks connector The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md.

Authentication errors Verify: - access token is valid and not expired - HTTP path is correct for your SQL warehouse - workspace hostname is correct

See Also

- [[drivers]]
- **db**
- **validate**
- [[odbc]]
- INSTALL.md

Greenplum (Python Driver)

Overview

TrueBench driver name (used in the DB command): `greenplum`

This driver connects to VMware Greenplum using the PostgreSQL wire protocol and a Python PostgreSQL connector.

If you specifically need ODBC behavior or must match other ODBC tools, use [[odbc]] with an appropriate PostgreSQL/Greenplum ODBC driver (if installed).

Installation Behavior

When you use `greenplum` in the `DB` command to define an alias used to connect to Greenplum, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- wrong host or port
- wrong database name
- authentication failure
- firewall / VPN restrictions blocking access

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- psql
- DBeaver / DataGrip / VS Code SQL tools

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example (recommended alias: `gp`):

```
db gp greenplum host=myhost port=5432 database=mydb user=myuser password=?
```

Notes: - Greenplum commonly uses port 5432 - `password=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

```
validate gp
```

This validates: - connectivity and authentication - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and Greenplum - fastest achievable query response time via this client running TrueBench

Common Connection Parameters

Common examples: - `host=<hostname>` - `port=5432` - `database=<dbname>` - `user=<user>` - `password=<pwd>` (or `password=?`) - `sslmode=<mode>`

Refer to Greenplum/PostgreSQL connector documentation for supported parameter options.

Setup Directory (Benchmark Reporting)

If you want to integrate TrueBench RunID with Greenplum-side reporting, see:

`setup/greenplum/README.md`

This directory may contain scripts and notes for integrating benchmark run metadata with Greenplum query logging.

Troubleshooting

Authentication errors Verify: - user/password - database name is correct - server permits client network access

Network failures Verify the host is reachable from this client environment. If psql cannot connect from the same machine, TrueBench will not be able to connect.

See Also

- [[drivers]]
- **db**
- **validate**
- [[odbc]]
- INSTALL.md

IBM Db2 / Db2 Warehouse (Python Driver)

Overview

TrueBench driver name (used in the DB command): `ibm_db2`

This driver connects to IBM Db2 and Db2 Warehouse using the IBM Db2 Python driver (`ibm_db`).

IBM Integrated Analytics System (IIAS) is essentially a Db2 Warehouse appliance. There is no separate IIAS Python connector. IIAS uses the Db2 engine and protocols, so you connect using standard Db2 drivers and connection rules.

If you specifically need ODBC behavior or must match other ODBC tools, use [[odbc]] with an IBM Db2 ODBC driver (if installed).

Installation Behavior

When you use `ibm_db2` in the `DB` command to define an alias used to connect to Db2, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- wrong host / port
- wrong database name
- authentication failure
- SSL configuration requirements
- firewall / VPN restrictions blocking access

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- DBeaver / DataGrip / VS Code SQL tools (Db2 adapters)
- IBM Db2 client tools used in your organization

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example (recommended alias: `db2`):

```
db db2 ibm_db2 host=myhost port=50000 database=BLUDB user=myuser password=?
```

Notes: - default Db2 port is environment dependent - `password=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

```
validate db2
```

This validates: - connectivity and authentication - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and Db2 - fastest achievable query response time via this client running TrueBench

Common Connection Parameters

Common examples: - `host=<hostname>` - `port=<port>` - `database=<dbname>` - `user=<user>` - `password=<pwd>` (or `password=?`)

SSL and security settings vary widely across environments. Consult IBM Db2 documentation and your organization's security requirements for supported options.

Setup Directory (Benchmark Reporting)

If you want to integrate TrueBench RunID with Db2-side reporting, see:

```
setup/ibm_db2/README.md
```

This directory may contain scripts and notes for integrating benchmark run metadata with Db2 system logging.

Troubleshooting

Cannot import ibm_db The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md.

Authentication / SSL errors Verify: - host/port and database - user/password - SSL requirements and certificates (if required)

See Also

- [[drivers]]
- **db**
- **validate**
- [[odbc]]
- INSTALL.md

IBM Netezza Performance Server (Python Driver)

Overview

TrueBench driver name (used in the DB command): `ibm_netezza`

This driver connects to IBM Netezza Performance Server using the official `nzpy` Python driver. `nzpy` is a pure Python DB-API 2.0 connector maintained by IBM.

If you specifically need ODBC behavior or must match other ODBC tools, use [[odbc]] with a Netezza ODBC driver (if installed).

Installation Behavior

When you use `ibm_netezza` in the `DB` command to define an alias used to connect to Netezza, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- wrong host / port
- wrong database name
- authentication failure

- firewall / VPN restrictions blocking access

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- DBeaver / DataGrip / VS Code SQL tools (Netezza adapters)
- IBM Netezza client tools used in your organization

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example (recommended alias: `nz`):

```
db nz ibm_netezza host=myhost port=5480 database=mydb user=myuser password=?
```

Notes: - the default port may vary by environment - `password=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

```
validate nz
```

This validates: - connectivity and authentication - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and Netezza - fastest achievable query response time via this client running TrueBench

Common Connection Parameters

Common examples: - `host=<hostname>` - `port=<port>` - `database=<dbname>` - `user=<user>` - `password=<pwd>` (or `password=?`)

Depending on your environment, additional parameters may exist for SSL and network behavior. Consult IBM Netezza documentation for supported options.

Setup Directory (Benchmark Reporting)

If you want to integrate TrueBench RunID with Netezza-side reporting, see:

```
setup/ibm_netezza/README.md
```

This directory may contain scripts and notes for integrating benchmark run metadata with Netezza logging and history tables.

Troubleshooting

Cannot import nzpy The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md.

Authentication errors Verify: - user/password - database name is correct - server permits client network access

See Also

- [[drivers]]
- **db**
- **validate**
- [[odbc]]
- INSTALL.md

Generic ODBC Driver

Overview

TrueBench driver name (used in the DB command): `odbc`

The generic ODBC driver in TrueBench allows you to connect to **any data source** for which an ODBC driver is installed on your system.
This includes DBMS servers, Access databases, Excel files, CSV directories, text files, and many other sources supported by the ODBC environment.

This driver is useful when: - You want to run the **sql** command against a data source that TrueBench does not have a built-in driver for. - You want to extract data (for example, to generate parameter files). - You want to experiment or validate connectivity before building full support for a DBMS in TrueBench.

Quick Setup Checklist (Recommended Order)

1) Confirm your ODBC driver is installed in the operating system
2) Use **list** to confirm the exact driver name
3) Define the alias in TrueBench using **db**
4) Run **validate** on the alias to ensure alias works and measure latency to the DBMS
5) Run **sql** statements using the alias

1) Confirm ODBC Driver Installed

TrueBench can only use ODBC drivers that are already installed and registered in the operating system.

ODBC drivers are NOT installed by TrueBench. They must be installed using OS-level installation packages provided by the DBMS vendor or Microsoft.

Basic OS notes:

- Windows: drivers are registered in the ODBC subsystem (visible in ODBC Data Sources)
- macOS: driver installs typically update `/Library/ODBC/odbcinst.ini`
- Linux: driver installs typically update `/etc/odbcinst.ini` (and require unixODBC)

If the ODBC driver is not installed correctly, TrueBench cannot use it.

2) Use LIST ODBC to Confirm Exact Driver Name

Use:

```
list odbc
```

TrueBench displays the installed driver names exactly as the OS reports them.
When defining an alias, you must use the **full driver name string**, including any parentheses ( ) and wildcard patterns *.

Example output:

```
truebench: list odbc
Installed ODBC drivers:
SQL Server
Teradata Database ODBC Driver 20.00
Microsoft Access Driver (*.mdb, *.accdb)
Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)
Microsoft Access Text Driver (*.txt, *.csv)
Microsoft Access dBASE Driver (*.dbf, *.ndx, *.mdx)
```

3) Define the Alias in TrueBench (DB Command)

Use the **db** command in this form:

```
db <alias> odbc driver="<ODBC driver name>" <param>=<value> ...
```

Every parameter after the driver name is copied directly into the ODBC connection string.
TrueBench does NOT validate whether parameters are correct for your specific driver.

Because each ODBC driver vendor uses different keywords, consult the driver documentation for required parameters.

Examples of common connection-string parameters (varies by driver): - `DBQ=` - `Server=` or `HOST=` - `Database=` - `UID=` / `PWD=` - `Trusted_Connection=Yes` - `ReadOnly=1`

4) Run VALIDATE (Required for Benchmark Credibility)

After defining an alias, always run:

```
validate <alias>
```

This ensures: - the alias works and the driver can connect successfully - TrueBench can execute SQL through the full ODBC stack - you have a baseline measurement for latency / overhead

`validate` measures: - round-trip latency through TrueBench + pyodbc + ODBC driver - fastest achievable response time from this client

This baseline is critical for benchmarking because: - for DBMS connections it identifies network latency and jitter, separate from DBMS execution time - when comparing two DBMS platforms, it acts as a fairness check (unequal network overheads can produce misleading results) - for local file-based ODBC sources (Excel/Access/CSV), it validates that the driver is working and measures local driver overhead

5) Run SQL Statements (Examples)

Once validated, use **sql** normally.

General example:

```
sql <alias> select 1
```

Example: Excel Workbook

Example DB alias:

```
db x odbc driver="Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)"
DBQ="C:/path/to/test.xlsx" ReadOnly=0 HDR=YES
```

Validate:

```
validate x
```

Example SQL (sheet names must be enclosed in [ ] and end with $):

```
sql x select * from [Sheet1$]
```

Notes: - Worksheet names must end with $ - The Excel driver requires a valid DBQ pointing to a workbook file - Consult Microsoft documentation for Excel ODBC options such as HDR or IMEX

Example: CSV or Text Files

Example DB alias:

```
db csv odbc driver="Microsoft Access Text Driver (*.txt, *.csv)"
DBQ="C:/path/to/folder"
```

Validate:

```
validate csv
```

Example SQL against a CSV file:

```
sql csv select * from sample.csv
```

Notes: - DBQ must refer to a directory, not a file - Column types and headers are inferred by the underlying ODBC driver

Critical Recommendations (Read This If You Get Connection Failures)

The generic ODBC driver can connect to almost anything, but TrueBench cannot know which parameters your ODBC driver requires.

Connection failures are common unless you consult the documentation for your specific driver. Common causes include:

- parameter names differ by vendor (for example DBQ vs Database vs SERVER vs Host)
- wrong server / user / password values
- firewall / VPN / network restrictions blocking the DBMS connection
- DSN-based driver requires `DSN=<name>` rather than Server/HOST parameters

Strong recommendation: - validate connectivity first with a trusted tool on the SAME machine where TrueBench runs

Debugging Connection Problems

If a connection fails, TrueBench displays: - the sanitized ODBC connection string - the ODBC driver name - the full ODBC error text (SQLSTATE codes, driver messages)

Common problems include: - misspelled driver names (must match exactly what `list odbc` shows) - missing Server/HOST parameters for DBMS drivers - incorrect DBQ path for Excel or Access - authentication failures - DSN-based drivers requiring `DSN=<name>` instead of `Server=`

Refer to your ODBC driver's official documentation for supported parameter names and required fields.

See Also

- [[drivers]]
- **list** odbc — displays installed ODBC drivers
- **db** — define and configure database connections
- **validate** — verify connectivity and measure baseline performance
- **sql** — execute SQL statements using any TrueBench connection

Oracle Database (Python Driver)

Overview

TrueBench driver name (used in the DB command): `oracle`

This driver connects to Oracle Database using the Oracle Python driver (`oracledb`).

Oracle Exadata is a deployment platform for Oracle Database. Exadata does not require a separate driver. If you can connect to Oracle, you can connect to Exadata.

If you specifically need ODBC behavior or must match other ODBC tools, use [[odbc]] with an Oracle ODBC driver (if installed).

Installation Behavior

When you use `oracle` in the `DB` command to define an alias used to connect to Oracle, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- wrong DSN / service name
- wrong host or port
- authentication failure
- SSL / wallet requirements (especially in cloud environments)
- firewall / VPN restrictions blocking access

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- SQL Developer
- SQLcl
- DBeaver / DataGrip / VS Code SQL tools (Oracle adapters)

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example (recommended alias: `ora`):

```
db ora oracle host=myhost port=1521 service_name=orclpdb1 user=myuser password=?
```

Notes: - Oracle connections often require a service name - `password=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

`validate ora`

This validates: - connectivity and authentication - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and Oracle - fastest achievable query response time via this client running TrueBench

Common Connection Parameters

Common examples: - `host=<hostname>` - `port=1521` - `service_name=<service>` - `user=<user>` - `password=<pwd>` (or `password=?`)

Some Oracle environments require wallets, thick-mode client installs, or additional SSL configuration. Consult Oracle and your organization's security guidance for supported options.

Setup Directory (Benchmark Reporting)

If you want to integrate TrueBench RunID with Oracle-side reporting, see:

`setup/oracle/README.md`

This directory may contain scripts and notes for integrating benchmark run metadata with Oracle logging and monitoring views.

Troubleshooting

Cannot import oracledb The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md.

DSN / service errors Verify: - service name is correct - you are using the correct host/port and listener configuration

Authentication / SSL errors Verify: - user/password - SSL / wallet requirements (if applicable)

See Also

- [[drivers]]
- **db**
- **validate**
- [[odbc]]
- INSTALL.md

PYDBAPI — Generic Python DB-API 2.0 Driver

Overview

The `pydbapi` driver enables TrueBench to connect to **any Python DB-API 2.0 compatible** database module.

This driver is intended for:

- new or niche database platforms not yet directly supported by TrueBench
- internal/vendor Python connectors where you can install the module into `.venv`
- rapid experimentation without changing TrueBench source code

The driver requires that the specified Python module exposes:

- `connect(...)`
- DB-API connection and cursor semantics (PEP 249)

TrueBench does **not** attempt to validate parameter names for DB-API modules. Any connection errors are handled consistently when the alias is first used.

Syntax

The `DB` command for `pydbapi` is:

```
DB <alias> pydbapi url=<endpoint> module=<python_module> [user=<user>]
[password=<pwd|?>] [param=value ...]
```

Notes:

- `module=<python_module>` is required
- `url=<endpoint>` is passed through to the module in one of two forms:
  - as a positional connection string (DSN), or
  - as a keyword argument (depending on the module)
- additional `param=value` settings are passed through to the module connect call

Required Parameters

- **module** — Python module name to import. Examples:
  - `teradatasql`
  - `psycopg`
  - `snowflake.connector`
  - `oracledb`
- **url** — endpoint identifier for the database. This may be a hostname, DSN, JDBC-like string, or platform-specific endpoint.

TrueBench uses `url=` for all drivers for consistency. The underlying module may interpret the value as a hostname, account identifier, or DSN depending on the DBMS.

Optional Parameters

- **user** — user name for the DBMS logon (optional for some DBMS platforms).
- **password** — password or `?` to prompt securely.
- **param=value** — any additional driver-specific connection parameters.

All additional parameters are passed through to the DB-API module `connect()` call.

Examples

Example 1 — Use Teradata SQL Driver as a Generic DB-API driver

```
DB tdg pydbapi url=de110922v174-3y0f2208aq7p4ckv.env.clearscape.teradata.com
module=teradatasql user=demo_user password=?
```

Example 2 — psycopg (PostgreSQL protocol)

```
DB pg pydbapi url=myhost module=psycopg user=myuser password=? database=mydb
port=5432
```

Example 3 — Driver requires DSN / connect string

Some DB-API modules accept only a positional DSN string (or accept it more reliably).

TrueBench attempts both forms automatically.

Example:

```
DB db2 pydbapi
url="DATABASE=BLUDB;HOSTNAME=db2host;PORT=50000;PROTOCOL=TCPIP;"
module=ibm_db user=db2inst1 password=?
```

Example 4 — Module with token auth (password used as token)

```
DB cloud pydbapi url=myacct module=snowflake.connector user=myuser password=?
warehouse=DEMO_WH role=ANALYST
```

Installation

The `pydbapi` driver does not auto-install arbitrary Python modules.

You must install the required Python module into the TrueBench `.venv`:

Linux/macOS: `./truebench.sh`

Windows: `truebench.bat`

Then install the module:

```
pip install <package>
```

Examples: - `pip install teradatasql` - `pip install psycopg[binary]` - `pip install ibm_db` - `pip install oracledb`

See **INSTALL.md** for installation details.

Notes / Limitations

- `pydbapi` is intended for DB-API 2.0 drivers.
- Some modules require very specific parameter names (example: `account`, `host`, `dsn`, `dbname`, etc). Use the vendor documentation.
- The TrueBench `PREPARE` command is not supported for the generic driver.
- Variable substitution (`:1`, `:2`, etc.) happens in TrueBench before the DBMS sees SQL.
- Some DB-API modules do not support per-statement cancel or per-statement timeout.

See Also

- [[drivers]]
- **db**
- [[odbc]]
- INSTALL.md

Amazon Redshift (Python Driver)

Overview

TrueBench driver name (used in the DB command): `redshift`

This is the preferred Redshift driver for TrueBench. It uses the Amazon Redshift connector for Python (module commonly imported as `redshift_connector`) and does not require any OS-level ODBC driver.

If you specifically need ODBC behavior or must match other ODBC tools, use [[odbc]] with a Redshift ODBC driver (if installed in your environment).

Installation Behavior

When you use `redshift` in the `DB` command to define an alias used to connect to Amazon Redshift, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

The common Python package name is:

- `redshift_connector`

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- incorrect cluster endpoint hostname
- wrong port (Redshift default port is typically 5439)
- wrong database name / user / password
- security group / firewall rules blocking connection
- private network restrictions (VPC-only clusters)

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- AWS Console: Redshift Query Editor v2
- psql (PostgreSQL client)
- DBeaver / DataGrip / VS Code SQL tools

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example (recommended alias: `rs`):

```
db rs redshift host=mycluster.abc123.us-east-1.redshift.amazonaws.com
port=5439 database=dev user=myuser password=?
```

Notes: - `host=` is the Redshift cluster endpoint hostname - `port=` defaults to 5439 if not specified (depends on your environment) - `password=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

```
validate rs
```

This validates: - endpoint correctness and connectivity to the Redshift cluster - authentication and login - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and Redshift - fastest achievable query response time via this client running TrueBench

This baseline helps separate performance impacts that are NOT DBMS execution time (especially for sub-second tactical queries), such as network latency and jitter.

It also provides a fairness check: when comparing two DBMS platforms, ensure the network overheads are equivalent across platforms — otherwise benchmark results may be misleading.

Important Benchmark Note: Redshift Result Cache

Amazon Redshift caches the results of certain types of queries. If an identical query is re-run and the cached result remains valid, Redshift can return cached results rather than executing the query.

For benchmarking, you may want to disable result caching for the current session.

Redshift provides a session parameter:

- `enable_result_cache_for_session`

To disable cached results for the current session:

```
SET enable_result_cache_for_session TO off;
```

If you leave result caching enabled, repeated benchmark runs can become dominated by cache reuse behavior rather than DBMS execution capability. :contentReferenceoaicite:1

Other Benchmark Notes

Redshift benchmark behavior can be influenced by factors unrelated to the SQL text itself:

- WLM queue configuration and concurrency limits
- cluster scaling and workload contention
- compilation and metadata caching effects
- table design (distribution style, sort keys) and data skew

Strong recommendation: - run multiple iterations - report distribution statistics (p50 / p95 / max), not only averages - avoid comparing single-run timing numbers

Common Connection Parameters

You may append additional connection parameters as `name=value` pairs. These are passed through to the Redshift Python connector.

Common examples: - `host=<cluster endpoint>` - `port=5439` - `database=<dbname>` - `user=<user>` - `password=<pwd>` (or `password=?`)

Common environment-dependent items include: - SSL requirements - IAM authentication or SSO integration (if enabled in your org)

If your organization uses IAM/SSO auth for Redshift, consult internal platform guidance for the correct connection method and supported parameters.

Setup Directory (Benchmark Reporting)

Many benchmarking teams maintain a host-side TestTracking table inside the benchmarked DBMS to support reporting and correlation by RunID (not just timestamps).

TrueBench can optionally maintain a host-side TestTracking table on your Redshift platform. This enables reporting workflows that join benchmark RunID to system telemetry.

If you want to integrate TrueBench RunID with Redshift-side reporting, see:

`setup/redshift/README.md`

This directory contains scripts to create: - a host-side TestTracking table (optional) - reporting functions/views joining Redshift telemetry to runid

Troubleshooting

Cannot import redshift_connector The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md.

Authentication errors Verify: - user/password - DB name is correct - your account is allowed to connect (Redshift permissions)

Network failures Verify the cluster endpoint is reachable from this client environment and that the Redshift security group allows inbound traffic from the client (IP/VPC rules).

If a familiar Redshift client tool cannot connect from the same machine, TrueBench will not be able to connect.

See Also

- [[drivers]]
- **db**
- **validate**
- [[odbc]]
- INSTALL.md

Snowflake (Python Driver)

Overview

TrueBench driver name (used in the DB command): `snowflake`

This is the preferred Snowflake driver for TrueBench. It uses the Snowflake Connector for Python (module: `snowflake.connector`) and does not require any OS-level ODBC driver. :contentReferenceoaicite:0

If you specifically need ODBC behavior or must match other ODBC tools, use [[odbc]] with a Snowflake ODBC driver (if installed).

Installation Behavior

When you use `snowflake` in the `DB` command to define an alias used to connect to Snowflake, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed

2. Offer to install the required pip package(s) into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

The Python package typically installed is:

- `snowflake-connector-python` :contentReferenceoaicite:1

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- incorrect Snowflake account identifier
- wrong user / password (or SSO / external browser auth requirements)
- wrong role / warehouse / database / schema
- corporate firewall / proxy restrictions blocking outbound access

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust, for example:

- Snowflake Snowsight (Web UI)
- Snowflake CLI
- SnowSQL (where available in your environment)
- DBeaver / DataGrip / VS Code SQL tools

If those tools cannot connect from the same machine, TrueBench will not be able to connect either. :contentReferenceoaicite:2

Defining a DB Alias

Example (recommended alias: `sf`):

```
db sf snowflake account=myacct user=myuser password=? warehouse=COMPUTE_WH
database=MYDB schema=PUBLIC role=ANALYST
```

Notes: - `account=` is your Snowflake account identifier - `warehouse=` is required for query execution - `password=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

```
validate sf
```

This validates: - account correctness and connectivity to Snowflake - authentication and session creation - warehouse access and basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and Snowflake - fastest achievable query response time via this client running TrueBench

This baseline helps separate performance impacts that are NOT DBMS execution time (especially for sub-second tactical queries), such as network latency and jitter.

It also provides a fairness check: when comparing two DBMS platforms, ensure the network overheads are equivalent across platforms — otherwise benchmark results may be misleading.

Important Benchmark Note: Snowflake Result Cache

Snowflake can reuse persisted query results for identical queries. This can produce extremely fast query times that do NOT reflect actual execution performance.

By default, result reuse is enabled, but it can be overridden using the session parameter:

- `USE_CACHED_RESULT`

For benchmarking, you may want to disable cached result reuse to ensure each query truly executes on the DBMS:

- `USE_CACHED_RESULT=false` :contentReferenceoaicite:3

If you leave cached results enabled, repeated benchmark runs can become dominated by cache behavior rather than DBMS execution capability.

Common Connection Parameters

You may append additional connection parameters as `name=value` pairs. These are passed through to the Snowflake Python connector. :contentReferenceoaicite:4

Common examples: - `account=<acct_identifier>` - `user=<user>` - `password=<pwd>` (or `password=?`) - `warehouse=<warehouse>` - `database=<dbname>` - `schema=<schema>` - `role=<role>`

Refer to Snowflake documentation for supported connector parameters and configuration. :contentReferenceoaicite:5

Setup Directory (Benchmark Reporting)

Snowflake provides strong native query history and warehouse usage telemetry. In addition, many benchmarking teams maintain a host-side TestTracking table inside the benchmarked DBMS to support automated reporting and correlation by RunID (not just timestamps).

TrueBench can optionally maintain a host-side TestTracking table on your Snowflake platform to enable reporting workflows that join benchmark RunID to Snowflake query history.

If you want to integrate TrueBench RunID with Snowflake-side reporting, see:

`setup/snowflake/README.md`

This directory contains scripts to create: - a host-side TestTracking table (optional) - reporting functions/views joining Snowflake usage/history to runid

Troubleshooting

Cannot import snowflake.connector The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md. :contentReferenceoaicite:6

Authentication errors Verify: - user/password or SSO requirements - role access - corporate VPN / MFA requirements

Warehouse errors Most Snowflake query execution requires a valid warehouse. Verify: - warehouse exists - you have USAGE permission - warehouse is not suspended (or can auto-resume)

Unexpectedly fast results Likely caused by Snowflake persisted query results. Disable cached result reuse using `USE_CACHED_RESULT=false`. :contentReferenceoaicite:7

See Also

- [[drivers]]
- **db**
- **validate**
- [[odbc]]
- INSTALL.md

SQLite (Built-in Driver)

Overview

TrueBench driver name (used in the DB command): `sqlite3`

SQLite is a built-in local database engine. It is always available in TrueBench and requires: - no installation - no URL / hostname - no user / password

SQLite is used by TrueBench internally for metadata tracking (TestTracking / TestResults), and can also be used as a convenient local DBMS for development and testing.

For SQLite SQL language reference and examples, see: - **sqlite**

Defining a DB Alias

Example (recommended alias: `meta`):

```
db meta sqlite3 file=metadata/truebench.db
```

Notes: - `file=` is a local path (relative or absolute) - if the file does not exist, SQLite will create it

Validate Connectivity

After defining an alias, run:

```
validate meta
```

This confirms the file can be opened and basic SQL execution works.

See Also

- [[drivers]]
- **db**
- **validate**
- **sqlite**

Teradata (Python Driver)

Overview

TrueBench driver name (used in the DB command): `teradata`

This is the preferred Teradata driver for TrueBench. It uses the Teradata SQL Driver for Python (A.K.A. `teradatasql`) and does not require any OS-level ODBC driver.

If you specifically need ODBC behavior or must match other ODBC tools, use [[teradata_odbc]].

Installation Behavior

When you use `teradata` in the `DB` command to define an alias used to connect to Teradata Vantage, if the required Python module is missing, TrueBench will:

1. Inform you the driver module is not installed
2. Offer to install the required pip package into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

Before You Benchmark (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- wrong hostname / URL
- wrong user / password (or authentication mechanism)
- firewall / VPN / network restrictions blocking the DBMS connection

To avoid wasting time troubleshooting inside a benchmark tool, it is strongly recommended that you first connect successfully from the SAME machine where TrueBench is running using a tool you already trust (for example: Teradata Studio / SQL Assistant / BTEQ / DBeaver).

If that tool cannot connect, TrueBench will not be able to connect either.

Defining a DB Alias

Example:

db td teradata url=myhost user=myuser password=?

Notes: - `url=` is the Teradata host / system name / IP address - `password=?` prompts securely

Validate Connectivity (Do this next)

After defining an alias, always run:

validate td

This validates: - URL correctness and connectivity to the Teradata system - authentication and login - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and the Teradata server - fastest achievable query response time via this client running TrueBench

This baseline helps separate performance impacts that are NOT DBMS execution time (especially for sub-second tactical queries), such as network latency and jitter.

It also provides a fairness check: when comparing two DBMS platforms, ensure the network overheads are equivalent across platforms — otherwise benchmark results may be misleading.

Common Connection Parameters

You may append additional connection parameters as `name=value` pairs. These are passed through to `teradatasql.connect(...)`.

Common examples: - `encryptdata=true` (default used by TrueBench if not specified) - `logmech=TD2|LDAP|JWT` - `database=<dbname>` - `charset=UTF8`

Refer to Teradata documentation by searching for: “Teradata SQL Driver for Python” for the full supported parameter set.

Setup Directory (Benchmark Reporting)

TrueBench includes the ability to maintain a TestTracking table on your Teradata platform. This allows you to join that table to query logging (DBQL) and resource usage tables (ResUsage) to analyze why you got the performance that was observed during tests.

By joining the TestTracking table to the system logging tables, you can analyze system information by RunID instead of tracking and using timestamps.

If you want to join TrueBench `runid` to Teradata query logging (DBQL), see:

setup/teradata/README.md

This directory contains scripts to create: - a host-side TestTracking table (optional) - reporting functions/views joining DBQL to runid

Troubleshooting

Cannot import teradatasql The module is not installed. Re-run the DB command and accept the install prompt, or install manually as described in INSTALL.md.

Authentication errors Verify: - user/password - logmech - corporate VPN / MFA requirements

Network failures Verify the host in `url=` is reachable from this client environment. If a familiar Teradata client tool cannot connect from the same machine, TrueBench will not be able to connect.

See Also

- [[drivers]]
- **db**
- **validate**
- [[teradata_odbc]]
- INSTALL.md

Teradata (ODBC Driver)

Overview

TrueBench driver name (used in the DB command): `teradata_odbc`

TrueBench can connect to Teradata using ODBC. Use this driver only when you specifically need ODBC behavior (for example, to match other ODBC-based tools).

The recommended Teradata driver for TrueBench is **teradata** (Python driver). It is preferred because:

- TrueBench can offer to install the required Python module automatically
- ODBC requires OS-level driver installation (manual / admin-controlled)

- the Python driver is often materially faster for tactical queries

Requirements - Python package: `pyodbc` - OS-level Teradata ODBC driver installed for your platform

TrueBench can install `pyodbc` automatically, but the Teradata ODBC driver must be installed manually.

Quick Setup Checklist (Recommended Order)

Because ODBC setup requires both Python and OS components, use this order:

1. Install the Teradata ODBC driver in the OS (Windows/macOS/Linux)
2. Confirm the driver works using an ODBC-capable tool (before TrueBench)
3. Define the alias in TrueBench:

   db td teradata_odbc url=myhost user=myuser password=?

4. Ensure the alias works and measure latency to the DBMS with:

   `validate td`

Installation Behavior

When you use `teradata_odbc` in the `DB` command to define an alias used to connect to Teradata Vantage, if the required Python module is missing, TrueBench will:

1. Inform you that `pyodbc` is not installed
2. Offer to install the required pip package into the TrueBench .venv
3. Proceed if installation succeeds

If installation is declined or blocked (firewall / restricted pip), the alias is not created. You will then need to review INSTALL.md for manual installation instructions.

If the OS-level Teradata ODBC driver is missing, TrueBench cannot create the alias.

What pyodbc Does

`pyodbc` is the Python interface that allows TrueBench to communicate with ODBC drivers. It is required for all ODBC-based DBMS connections.

Once installed, TrueBench can detect any compatible ODBC driver available on the system.

What the Teradata ODBC Driver Does

The Teradata ODBC driver is provided by Teradata and must be installed separately. It allows TrueBench and other ODBC applications to open sessions and execute SQL on Teradata.

Driver packages depend on your operating system:

- Windows uses MSI installers
- macOS uses PKG installers
- Linux uses RPM or DEB packages

Where To Download the Teradata ODBC Driver

Search for:

Teradata ODBC Driver download

Select **Teradata Tools and Utilities**, then choose:

Client Tools → ODBC Driver

Download the version that matches your operating system.

Windows Installation Details

The Windows download is typically a ZIP file containing several components.

After downloading:

1. Extract the ZIP to a local directory
2. Locate the file:

   SuiteSetup.exe

3. Double-click `SuiteSetup.exe` to install the driver

This is the correct installer for Windows.

Note: the batch files (such as `silent_install.bat`) do not install the driver unless invoked with a full feature list. Running them without arguments may only display usage information.

After installation, TrueBench should detect the driver automatically. You can confirm with:

list drivers

A checkmark under the ODBC column for the `teradata_odbc` driver indicates detection.

macOS Notes

Install the PKG installer appropriate for your CPU architecture (Intel or Apple Silicon).

After installation, a Teradata entry should appear in:

/Library/ODBC/odbcinst.ini

If TrueBench does not detect the driver, verify the PKG installer completed successfully.

Linux Notes

Linux distributions require platform-specific packages:

- RPM for RHEL, SUSE, Rocky, Alma
- DEB for Ubuntu or Debian

After installation, confirm that a driver entry appears in:

/etc/odbcinst.ini

Your system also requires the `unixODBC` package. Some distributions install it automatically; others require:

sudo apt install unixodbc or sudo yum install unixODBC

If TrueBench cannot locate the driver, verify that the ODBC configuration files exist and that permissions allow them to be read.

Before You Use TrueBench (Strong Recommendation)

Most connection problems are NOT related to TrueBench. The most common issues are:

- wrong hostname / URL
- wrong user / password (or authentication mechanism)
- firewall / VPN / network restrictions blocking the DBMS connection
- missing / misconfigured OS-level ODBC driver

Before attempting to use ODBC inside TrueBench:

1) Verify the Teradata ODBC driver is installed
2) Verify it is registered in the OS ODBC configuration
3) Verify that a trusted tool can connect from the same machine

Examples of tools that can validate the ODBC driver installation:

- Teradata Studio / SQL Assistant
- DBeaver (ODBC connection)
- OS-specific ODBC administrator / DSN configuration tools

If those tools cannot connect using ODBC from the same machine, TrueBench will not be able to connect either.

Defining a DB Alias

Example:

```
db td teradata_odbc url=myhost user=myuser password=?
```

Notes: - `url=` is the Teradata host / system name / IP address - `password=?` prompts securely

Common Connection Parameters

You may append additional connection parameters as `name=value` pairs. These are passed through to the ODBC connection.

Common examples: - `charset=UTF8` - `database=<dbname>` - `sessionmode=ANSI|TERADATA` - `logintimeout=<seconds>` - `querytimeout=<seconds>` - `mechanism=ldap|krb5|jwt` - `sslenabled=true` - `ssltruststore=<path>`

For the full supported parameter set, search Teradata documentation for:

“Teradata ODBC Driver for Teradata Tools and Utilities — User Guide”

Validate Connectivity (Do this next)

After defining an alias, always run:

validate td

This validates: - URL correctness and connectivity to the Teradata system - authentication and login - basic query execution

In addition, `validate` provides a performance baseline by measuring: - round-trip latency between the TrueBench client and the Teradata server - fastest achievable query response time via this client running TrueBench

This baseline helps separate performance impacts that are NOT DBMS execution time (especially for sub-second tactical queries), such as network latency and jitter.

It also provides a fairness check: when comparing two DBMS platforms, ensure the network overheads are equivalent across platforms — otherwise benchmark results may be misleading.

Recommendation If you are uncertain whether to use ODBC, define BOTH drivers and compare:

```
db td_py teradata url=myhost user=myuser password=?
validate td_py
```

```
db td_odbc teradata_odbc url=myhost user=myuser password=?
validate td_odbc
```

If the Python driver shows lower latency and/or higher throughput, prefer it unless you must use ODBC to match other client tools.

Setup Directory (Benchmark Reporting)

TrueBench includes the ability to maintain a TestTracking table on your Teradata platform. This allows you to join that table to query logging (DBQL) and resource usage tables (ResUsage) to analyze why you got the performance that was observed during tests.

By joining the TestTracking table to the system logging tables, you can analyze system information by RunID instead of tracking and using timestamps.

If you want to join TrueBench `runid` to Teradata query logging (DBQL), see:

`setup/teradata/README.md`

This directory contains scripts to create: - a host-side TestTracking table (optional) - reporting functions/views joining DBQL to runid

Troubleshooting

Missing ODBC Driver If TrueBench reports:

Alias not created because its ODBC driver is missing.

Then the Teradata ODBC driver is not installed or not detectable. Reinstall the driver and try defining the alias again.

Missing pyodbc If TrueBench reports:

Alias not created because pyodbc is required.

Re-run the DB command and accept the install prompt, or install manually:

```
pip install pyodbc
```

IM002 errors (validate) Error IM002 indicates the ODBC driver is missing, corrupted, or not registered in the system's ODBC configuration. Reinstalling the Teradata ODBC driver usually resolves this.

See Also

- [[drivers]]
- **db**
- **validate**
- **teradata**
- INSTALL.md

Architecture of TrueBench

QueryDriver CLI Engine Architecture Summary (Checkpoint)

This topic summarizes the core architecture of the CLI execution engine, independent of the TUI layer.
It captures the control flow, parsing model, state model, worker orchestration, and logging pipeline.

1. Core Components

1.1 CommandProcessor

The central dispatcher and script engine.

Responsibilities: - Maintains authoritative GlobalState - Owns INCLUDE/EXEC input stack (`input_stack`) - Tokenizes, preprocesses, and executes commands - Performs variable expansion - Dispatches to per-command handlers via a `handlers` dictionary - Tracks `_raw_line` for commands that need the literal text (SQL/OS)

1.2 GlobalState

Persistent runtime model for a CLI session.

Key fields: - `db`, `db_lock_mode` - `runid`, `test_name`, `description` - `queues` and `workers` - Substitution variables (environment, system, user) - Output settings (`echo_mode`, `display_level`) - Execution control flags (`pause_event`, `kill_event`, `stop_flag`) - Status data for watchdog (`run_mode`, `duration_s`, `last_status`)

This is the authoritative execution context used by workers, logging, DB inserts, RunCommand, and the main loop.

1.3 InputSource (EXEC/INCLUDE Frames)

A stack frame representing an open script (`.tdb` file).

Each frame stores: - File handle or iterator - `include_vars` (for `:i1` substitution) - Saved echo mode - Indentation level for echo

`CommandProcessor.get_next_command()` always reads from the topmost frame until it is exhausted.

1.4 RunCommand

Encapsulates the implementation of the `RUN` command: - Parses flags (SERIAL, KILL, duration) - Validates queues and workers - Spawns worker threads - Spawns the watchdog (for timed runs and status updates) - Records status snapshots into `state.last_status` - Calls `finalize_run` at completion

1.5 Workers (workers.py)

Each worker thread: - Connects to the database - Pulls items from a queue via `queue.pop_next(...)` - Executes SQL or OS commands - Records success, error, and zero-row counts - Updates metadata (TestResults, row counts) - Responds to `stop_event`, `kill_event`, and timeouts

Workers are fully driven by CLI GlobalState.

2. Execution Phases

2.1 Parsing and Preprocessing

The CLI uses a unified preprocessing step:

```
line -> tokenize -> identify cmd -> substitution -> (cmd, args, raw_line)
```

This logic is encapsulated in:

```
processor.parse_next_command()
```

The logic handles: - Comment stripping - Blank lines - Tokenization via `shlex.split` - Selective substitution (for example, skip IF and control hooks when required) - Raw line preservation for SQL and OS commands

2.2 Script Execution Loop

This is the authoritative control loop (from `main.py`):

```
while True:
parsed = processor.parse_next_command()
if parsed is None: break
cmd, args, raw_line = parsed
processor._raw_line = raw_line
processor.execute(cmd, args)
```

This drives all `.tdb` scripts and ensures consistent behavior between: - CLI interactive entry - EXEC / INCLUDE stack processing - TUI-initiated script execution (when TUI calls the CLI engine)

2.3 Variable Expansion Model

QueryDriver supports multiple variable scopes:

1. Include variables `:i1 :i2`
    - Expanded only within include frames
2. Query parameters `:1 :2 ...` (from `.param` files)
3. Worker substitution values (for example per-worker identifiers)

4. System variables
    - `runid`, `testname`, `testdescription`, and related fields

5. Environment variables `${VARNAME}`

Substitution occurs in `parse_next_command` after command identification and before dispatch.

2.4 Queue and Worker Model

Define step
`DEFINE <testname> [description]`
- Clears queues, workers, and counters
- Starts a new DB session via `DatabaseManager.log_test_start()`

Queue step
`QUEUE q1 file=my.sql max_rows=5 ...`
- Creates queue definitions
- Parses queue configuration into objects stored in `state.queues`

Worker step
`WORKER q1 count=4`
- Allocates worker objects
- Stores worker definitions in `state.workers[qname]`

Run step
- Validates queues and workers
- Spawns worker threads
- Executes until exhaustion, time limit, external stop, or kill

3. Logging Pipeline

3.1 DatabaseManager.add_log

Every log line flows through the DB subsystem:

1. Insert into `TestTrackingLog` (when logging is enabled)

2. Print to console subject to log level and `echo_mode`

3. Forward to TUI Run Monitor (if active)

3.2 Log Types

Common log types include: - `INPUT` - `INFO` - `ERROR` - `SQL` - `OS` - Worker-level row count and summary logs

3.3 Log Filters

Console visibility is controlled by: - `interactive_input` (interactive vs script) - `echo_mode` - `display_level`

ERROR, SQL, and OS messages are always visible.
INPUT lines are visible depending on echo and interactive mode.

4. TestTracking and Results

The CLI writes to SQLite metadata tables.

TestTracking
Run-level metadata: - `RunId`
- `TestName`
- `Description`
- Start and end timestamps
- Summary statistics

TestTrackingLog
- Every log line recorded by CLI or workers (when logging is enabled)

TestResults
- Per-statement row counts and timing (when requested)

All DB inserts for results are flushed by a background flusher in `DatabaseManager` using a queue.

5. Error Handling

Parsing errors
- Reported via `processor.log("ERROR", ...)`
- Do not stop script execution unless they indicate a fatal condition

Command handler errors
- Logged and execution continues unless the error is explicitly fatal

Worker errors
- Worker increments its error counter
- Worker continues to the next item unless `stop_event` is set

Fatal DB errors
- Permanently disable logging to the metadata database
- Console logging continues so the user can still see diagnostics

6. Status and Watchdog

The RUN command spawns a watchdog thread that: - Calculates elapsed and remaining time for timed tests

- Emits periodic status messages
- Detects external pause, stop, and kill flag files
- Computes per-queue statistics (queries executed, errors)
- Updates `state.last_status` snapshot for external consumers

This snapshot can be used by TUI, future GUIs, or telemetry, but is driven entirely by the CLI engine.

7. CLI Interactive Mode

If no EXEC/INCLUDE frames exist, the CLI enters interactive prompt mode:

```
> user enters line -> logged as INPUT -> parse -> execute
```

When a script completes, control returns to interactive mode automatically.

The prompt is suppressed while scripts are running so worker and SQL output can appear cleanly.

8. Summary of CLI Guarantees

- Deterministic script execution
- Consistent preprocessing across CLI and TUI entry points
- Reliable worker parallelism and queue scheduling
- Centralized logging and metadata tracking
- Fully isolated execution state in GlobalState
- Predictable error behavior and recovery
- Extensible command set via a handler dictionary

This is the canonical, authoritative execution engine for QueryDriver's CLI layer.

QueryDriver TUI and CLI Integrated Architecture

This topic summarizes how the TUI layer cooperates with the CLI engine.
It describes execution flow, state separation, logging, and F5 test execution.

1. Global Architecture Overview

QueryDriver consists of two cooperating execution environments.

A) CLI Engine (authoritative execution path)

- Processes .tdb scripts
- Manages INCLUDE and EXEC input stack
- Performs tokenizing and substitution
- Dispatches commands to handlers
- Manages queues, workers, and run execution
- Inserts logs into SQLite metadata tables
- Feeds logs to the TUI when a TUI run is active

B) TUI Editor and Runner

- Provides a full screen curses editor
- Allows editing of testname, description, queues, workers
- Provides F2 file browse, F6 save, F5 run
- Displays help files using the TUI rich viewer
- During F5 test execution, the TUI does not run SQL directly.
  It generates a .tdb script and hands it to the CLI for execution.

2. Clean Separation of State

Two independent GlobalState objects are used.

CLI GlobalState

- Used only by the CLI execution engine
- Contains all information required for actual SQL execution
- Tracks workers, queues, DB connections, metadata logging
- Used by RUN command, workers, and watchdog

TUI GlobalState

- Used only by the TUI front end

- Stores editor fields, screen references, modal state
- Stores TUI selections such as which queue or worker is highlighted
- Does not contain any executable worker or queue objects
- Is never used to execute SQL

The TUI sends its state one way into the CLI by generating a script file.

3. F5 Execution Control Flow

Pressing F5 executes the test via the CLI engine.

Step 1. Build .tdb script

- The TUI collects the editor fields
- A helper builds a .tdb script from the TUI state
- The script is written to _temp_tui_script.tdb under the truebench home

Step 2. CLI signals

The CLI GlobalState receives:

- tui_run_active = True
- tui_run_screen = stdscr for log forwarding

The TUI and CLI remain strictly separated.

Step 3. Execute the script using EXEC

The TUI invokes:

```
processor.execute("EXEC", [temp_script])
```

This pushes the file onto the CLI input stack.

Step 4. Synchronous execution

The TUI now loops exactly as the CLI main loop does:

```
while True:
parsed = processor.parse_next_command()
if parsed is None:
break
cmd, args, raw = parsed
processor.execute(cmd, args)
```

This preserves all CLI semantics including raw_line and substitution rules.

Step 5. TUI waits until CLI finishes

The F5 key handler blocks until CLI processing completes.
After completion, TUI regains control and displays a completion message.

4. TUI Run Monitor Architecture

The code for the run monitor exists but is temporarily disabled until synchronous F5 operation is fully stabilized.

When active, the monitor will:

- Display testname, runid, elapsed time
- Show per-queue statistics from watchdog snapshots
- Show a scrolling log window populated by CLI log forwarding
- Allow stop or kill operations

Safety features already implemented:

- modal_active pauses the monitor refresh loop
- current_screen ensures msgboxes appear in correct windows
- tui_run_screen attaches and detaches cleanly

5. Modal Message Handling

The msgbox subsystem uses a robust modal design.

Key behaviors

- modal_active = True suspends TUI refresh loops
- safe_error_popup shows an error window centered on the screen
- The TUI main loop and any monitor loops pause during modal dialogs
- Return from msgbox restores previous modal state

6. Preprocessing and Command Execution

Both CLI and TUI execution paths use the same command preprocessing.

parse_next_command performs

- Retrieve a line from input stack or prompt
- Skip blank lines and comments
- Tokenize with shlex
- Identify command prior to substitution
- Apply expand_vars except for restricted commands
- Preserve raw_line for SQL or OS commands
- Return cmd, args, raw_line

This guarantees identical behavior for interactive CLI, EXEC scripts, and TUI-initiated scripts.

7. Logging and Stdout Redirection

DatabaseManager.add_log performs

- Insert into SQLite TestTrackingLog
- Print to stdout if interactive
- Forward logs to the TUI run window when tui_run_active is True

Stdout redirection

When TUI execution is active:

- sys.stdout is wrapped by TuiStdout
- All print() output goes to the TUI log area instead of the console

8. F5 Summary

- Build script from TUI
- Save to _temp_tui_script.tdb
- Execute via CLI EXEC synchronously
- TUI pauses until CLI completes

- Msgboxes appear safely even during run
- Logs flow into the TUI run window
- No asynchronous EXEC, no race conditions