Customer Best Practices for Teradata Benchmarking

Revision History

Summary of Changes			
Date	**Description of Change**	**Rev#**	**Author**
Aug 2010	First edition for publication	1.0	SFT Members
Sept 2011	Final draft for publication	2.0	Joanne Erceg, SFT Members
Feb 2012	Additional edits	2.1	Joanne Erceg

Important Notes	
Date	
Feb 2012	Final version

Table of Contents

Contents

1. Introduction

This paper is prepared for the use and benefit of Teradata customers. It is written by customers; for use by customers.

Managing data warehouse systems presents the challenge of change, which should be measured by a controlled means. Benchmarking is a best-practice method which can be used to measure performance, scalability and functionality. It provides a data-driven framework for such assessments rather than relying on other experiences that may be anecdotal, less scientific or difficult to reproduce once a major change has already been deployed, such a system expansion or major release software upgrade.

Benchmark results can help to answer questions such as: 1) Are we getting what we paid for? 2) What potential risks do we have of hitting bugs or regressions? 3) Is my workload designed to take advantage of performance-related features and/or changes to the environment that increase linear scalable performance boosts?

For example:

- **The investment of a system floor sweep or expansion is expected to deliver a range of specific linear scalable boost over the current configuration.** A test suite with queries and data that will scale across AMPs can measure both predictive scalable boost as well as potential regressions that may occur with dramatic hardware cost differences used by the optimizer.

- **A major release software upgrade may have an advertised out-of-box improvement.** The same test suite can be used to measure a predictive improvement that may be more accurate using queries and data from your actual workload. It may also uncover functional regressions and software feature failure during the pre-production test cycle.

- **A new feature and/or a data model change are being implemented.** A benchmark suite can be used to measure not only performance response times, but overall +/- impacts on CPU, I/O and resulting concurrency.

Building a performance benchmark test suite should be one of the first automation exercises in which to invest, from an operations point of view. A library of scalable tests can be executed before and after any change on the same system, or used to compare results between multiple systems and versions. This will help to cover risk by predicting performance and uncovering potential problems proactively. And while the idea of building a test suite may seem daunting or difficult, one can start simply and build up the suite over time.

Note to the reader. *This guide is provided as a courtesy of the Teradata Partners Service Focus Team (SFT). It comes as is; with no implied support. The procedures represent the combined learning of the customer members who contributed to it. The procedures are meant to be high level guides and do not represent step-by-step procedures. They are accurate* *to the best combined knowledge of the team, but should always be followed with caution and good judgment as they are not guaranteed to be free from flaws.*

2. Characteristics of a Robust Benchmark Suite

A robust benchmark suite is one that can run on different sized platforms and can scale in different situations. It should have the following characteristics;

- ✓ Queries:
  - Representative sample of actual workload
  - Queries should be a combination of tactical, short, medium, and long running queries
  - The same queries execute against the same data in the baseline and the comparison test
  - Execute in the same order
  - Execute with the same concurrency
    - Single stream
    - Multi-stream
  - Tag and number all queries in the SQL to identify it. If the same query is run multiple times, it can be identified as such in the results.
  - Use retlimit to limit the actual return row set unless response to client is to be included in the measurement or the rows themselves are going to be saved for checks on wrong results.
- ✓ Consistent logging:
  - DBQL on for all, including stepinfo
  - RSS settings on for all RSS subtables, and the same for all official runs.
- ✓ Test execution:
  - Tests are executed under the same conditions (e.g., quiesced database) every time.
  - Run directly BEFORE and AFTER the change to be measured.

    Note: Try to limit the number of major changes that are being measured in one test if at all possible. For example, if benchmarking for a technology refresh (floor sweep) where both the OS and Teradata software will also be a higher release than your current system, it would be prudent to run benchmarks on the OS change and Teradata version changes separately on a test system prior to such a production change. Discussions about upgrade strategies can be found in the SFT White Papers **Customer Best Practices for Teradata Upgrades – Major Software Upgrades and Major Hardware Upgrades.** Upgrading software on the source system to match the target system release prior to migrating would be the most ideal situation and could be benchmarked on production if this could be done in a separate step. Otherwise, it may be more difficult to ascertain the root cause for problems found.

  - Record all results on Teradata via DBQL or capture in another control table. EXPLAINS need to be captured for later reference.
  - If the test uses enabled TASM settings/workload definitions, ensure that the settings are the same between executions unless TASM features are the change being tested.

Adequate test coverage can be a challenge, depending upon how large and diversified the workload is. The goal is to build a suite that represents at least 25% of the most critical operational-ized workload (jobs that execute daily, weekly, etc.). A representative benchmark suite should include samples of all of the following types of workload:

- ✓ **All –AMP queries** – measures scalable improvement on hardware changes.
  a. Include a sample of 1 minute, 5 minute, 20 minute and longer complex queries

b. If your applications use objects with PPI, Join Index, partially covered indexes or rely heavily on certain coding constructs, these should be included in the benchmark

- ✓ **Tactical and short-running queries –** included for ensuring that they remain tactical/short running. If a workload contains thousands of short-running queries that suddenly, for example, stop using a join index and run 15 seconds each instead of 2 seconds, you would want to catch this regression early.

- ✓ **Inserts and deletes –** measures scalable hardware improvements and software changes involving possible feature changes (e.g., NUSI maintenance, WAL, etc.)

- ✓ **Samples from workloads with an SLA –** if these jobs are mission-critical, make sure they are represented in the test suite.

- ✓ **Samples from workloads that run only periodically** - i.e., month-end closing samples, anything that would have a serious business impact if failure or degradation should occur.

- ✓ **Choose/use tables that are large enough to scale** across the expected increase in number of AMPs, which is particularly important if measuring platform configuration changes.

- ✓ **Understand where your test data skew is** before and after changes are made that may change affect this. In an expansion or migration where the number of AMPs may increase substantially, skew in the data will get worse, and linear performance improvements may suffer because of it.

Extrapolating the risk coverage of a representative sample will depend on how large your system and workload are. The larger the sample, the more accurately predictive the results will be in terms of performance.

3. Considerations in Constructing a Benchmark Test Suite

Considerations in creating a benchmark suite should include the following:

1. How many queries do I need?
   a. The larger the variation in queries in the environment, the more you should include. Content coverage should answer the following:

      ✓What percentage of the workload is covered by your query set for performance or functionality?

      ✓Does the content include samples of high volume queries, such as short-running, quasi tactical for functional coverage?

   b. The goal is to cover as many basic variations as possible for the most important workloads. Queries can be sourced from DBQL and replayed as part of the benchmark suite.

2. What data will be used and where will it reside?

    a. Some customers use a sequestered database created specifically to hold benchmark data, and others use snapshots of real data on small test systems but the latest live data on production. This approach works as long as the "before" and "after" tests are done in a timely manner. A valid "before" test is run with days or hours before a change. A valid "after" test is run days or hours after a change.

    b. Arguments in favor of a "canned" benchmark data set are that you know exactly what is in it, and it can be designed to be moved to different platforms, provided that there is not too much disparity in the size of the target platforms, i.e., the data must scale across the larger number of AMPS, otherwise linear scalability cannot be accurately measured. Still, the smaller data set on the small system can be used to compare its own results to itself.

    c. Using "live" data works as long as the "before" and "after" tests are done in a timely manner. A valid "before" test is run within hours before a change. A valid "after" test is run within hours after a change.

    d. One point to remember is that a small dataset can be used on a small system but it may not scale properly on a large one. Still, the smaller data set on the small system can be used to compare its own results to itself.

    e. One practice for a benchmark with canned data is to keep scalable versions of the large tables on a test platform:

        1) The set of tables includes an alternative set of rows in a different database that are either smaller for small targets OR additional rows that can be added with insert/selects if the larger set is needed.

        2) The data, users, rights, etc. can be encapsulating for installation on demand when the suite is needed.

3. **Use TLE/TSET** to import costs from a prod system to a test system in order to simulate execution plans.

4. **Create a database repository of logging data** to where all DBQL and ResUsage data can be stored for a given software release. This is especially important when testing for performance across major software release upgrades. As long as the data is not in DBC, it will be preserved across any logging table changes, conversions or deletions resulting from Teradata software changes

5. **Create a database repository of benchmark results** for future reference. This is a more concise way of preserving results history than keeping many Excel spreadsheets alone.

6. **Script everything, including the retrieval of results** from the logging tables. This will ensure consistency of execution.

4. Benchmark Test Methods

For any test situation, there is a "first run" which serves as the baseline, and subsequent runs with changes to the environment that are compared to the baseline.

The test suite itself can be run from a network-connected driver node, using a series of shell or Perl scripts and bteq SQL files. Some customers also run their test suites from main frames or an actual TPA node (not the PDN). You can also use a network-connected PC, but be mindful that the PC itself does not become the bottleneck. A very good example of how to set this up on Windows can be found on the Teradata Developer Exchange, which includes a benchmark query driver.

Set up test users (for example: PerfTest01 – PerfTest20) who will be used to logon to the database and execute a pre-defined sequence of .sql files.

Single stream testing is the query set being executed sequentially, one at a time. Multi-stream can be done in different ways: 1) Run a set of predefined queries and mark how long it takes to complete, or 2) Run a set of predefined queries and mark how many complete in a predefined time limit, i.e., 1 hour. The reasons for running both single and multi-stream are that there could be changes being measured that affect concurrency which are not revealed in single stream, and there could be individual runtime degradation in the single stream which is worse in the multi-stream test results.

General steps in testing:

1. Ensure that the system is quiesced. An idle system should show all TPA nodes at 99+% IDLE.
2. Consider turning off syncscan in dbscontrol for the testing, particularly if the test is being conducted on a test system that is disparate in size/configuration compared to a larger production system.
3. Turn on all DBQL logging on all test IDs. If queries are tagged, limit query text to ensure logging is consistent across all query execution.
4. Ensure that RSS logging is turned on for all RSS sub-tables, use a finer logging/collection setting than you normally would (example: collect every 15 seconds, log every 30).
5. Mark the start times and end times of all tests programmatically with an insert into your performance results database.
6. Once the test is over, reinstate RSS settings and dbscontrol parameters.

5. Using Test Results

Using a predicted measurement like TPERF (a performance throughput number assigned by Teradata to configurations), if the baseline configuration is 1.0 and the target improvement is 2.5, the expected difference is that overall, any all-AMP queries with high parallel efficiency should be 2.5 x faster on the new target. This is the most simplistic way of measuring linear scalable boost when doing comparative testing between targets, such as in a tech-refresh/floor sweep, where you may be migrating from one platform to a larger/more powerful one, or in an expansion, where more nodes are being added to the same platform, thereby increasing the parallelism and overall power of the platform.

In the case of software changes alone, if anticipated ”out of box” performance boost is advertised, the underlying improvements may not be as obvious. In this context, any such performance boost attributable to software enhancements alone without changes to the workload and the reasons for such a boost should be quantifiable, particularly if you don't see this in the results.

In either case, the execution plans can change for a given query, so it is essential to collect all EXPLAINS for every test iteration. It may also be important to collect the stepinfo for all queries, as the length of actual time spent in each step can vital clues as to where a query has gone awry in terms of CPU path regression or other problems.

The main results of interest from DBQL are:

1. **Total execution time per query, and overall for each test**
   - The bottom line is how long each query took to execute. During a concurrency benchmark test, how long it took the entire suite to execute equates to a measure of performance boost or lack thereof.
   - Any individual queries that ran longer compared to the baseline should be investigated for plan degradation and the fix for the problem tracked as required.
2. **Total CPU and I/O consumption per query, and over all for each test**
   - Generally speaking, if a query used less CPU and less I/O than the baseline, one would expect it to run faster. Overall, less CPU and I/O consumption would also imply the ability to execute with higher concurrency. However, it is possible with a plan change to execute faster but with higher CPU, such as certain types of hash join versus merge join which can be seen in the execution plan.
3. **Number of steps in each query executed**
   - When lining up queries in a direct comparison between test runs, the number of steps can be an indicator of plan changes when used in conjunction with CPU and I/O numbers.
4. **Number of rows found (result rows).**
   - Assuming that neither the queries nor the test data sets have changed between before/after results, the number of rows found should be exactly the same. If they are not, then this should be investigated to understand why. Inconsistent or wrong results may return different numbers of rows, but this is still no guarantee that the same number of rows found are actually the same rows between runs, particularly if testing between different software versions with possibly different optimizer features enabled. If this is a concern, then the actual returned rows are needed. This can also be scripted to sanity check the output (e.g., diff output files).

5. **Overall parallel efficiency of the executed workload using the efficiency of each query executed.** Exclude single/few AMP queries in this calculation**.** Drops in query PE can indicate plan regression.

Since there is no direct linkage between physical and logical I/O at the query level, it is useful to turn on and retain **ResUsage** data for all runs. In the single stream test runs, it can illustrate the actual physical I/O impact of a query versus the logical I/Os. DBQL and AMPUsage I/Os are logical I/Os.

Analyzing Single Stream Results

The single stream results will capture the true standalone execution time of each query participating in the multi-stream benchmark. When comparing results to the baseline, if the plans look the same or the plans in the comparison actually look improved over the baseline but the runtime is longer, then use the stepinfo captured to determine which steps are taking longer. For software release testing, it is helpful to know in advance what features are 'new' and on by default, which can help explain differences in execution plans, both positive and negative.

Analyzing Multi-Stream Results

The purpose of the multi-stream benchmark is to see how the system handles concurrency. Some factors affecting results:

- How system busy the test suite drives the system – if the system is not 100% system busy, the comparison should still show linear scalability on an expansion or tech refresh where some boost is expected, but won't show how the system handles high concurrency bottlenecks
- I/O bottlenecks, including slow LUNs (marginally defective array drives that have not yet failed) or I/O constrained configurations
- Memory per AMP/node differences. Comparing larger memory configurations to smaller ones could show positive effects of increased data block caching capabilities unless dbscachethresh is set to zero in dbscontrol.

In general, any benchmark testing is always better than no testing at all. Anyone can start small and build up the suites over time. The important point to remember is that even if the reasons for a bad result are not glaringly obvious, if the logging data noted in this paper is saved for the baseline and test comparison runs, they can be investigated and recreated by Teradata for answers and resolutions.

6. Closing

The SFT is a committee of the PARTNERS User Group that works closely with Teradata on issues related to support services and other areas that fall beyond the scope of product enhancements. Members represent the concerns of Teradata customers by serving as catalysts for service improvements, providing ongoing feedback to Teradata and the PARTNERS Steering Committee.

The members of the SFT sincerely hope that you found this document useful. Please let us know of any comments or suggestions you may have. See you at Partners!

You can find our contact information and more about the SFT at our website:

http://www.teradata-partners.com/partners-user-group/service-focus-team